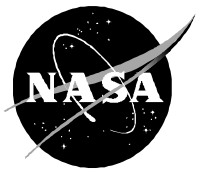
NASA Technical Paper 3634



Quantitative Measures for Software Independent Verification and Validation

Alice Lee, Project Technical Manager

December 1996

NASA Technical Paper 3634

Quantitative Measures for Software Independent Verification and Validation

Alice Lee, Project Technical Manager

Lyndon B. Johnson Space Center
Houston, Texas

December 1996

National Aeronautics
and Space Administration

Lyndon B Johnson Space Center
Houston, Texas 77058-4406

Foreword

The primary objective of this project is the formulation of software independent verification and validation (IV&V) methodologies that can be readily applied to future software development projects both for and within the National Aeronautics and Space Administration (NASA). Inherent in the methodologies is the precise specification of software measurement and data collection processes in the software development process so that more precise comparisons may be made in future software verification and validation activities. This objective will be achieved through the modification of existing reliability models to incorporate measures of dynamic program complexity. We now understand that reliability of a software system is directly determined by the execution environment of the software. Our goal is to build on the measurement methodologies established in the first phase of this project to improve the accuracy of software reliability assessment for the Space Shuttle primary avionics software system (PASS) and, ultimately, all software delivered to NASA.

Acknowledgments

This work was completed under the support of the Research and Technology Objectives and Plans (RTOP) funding, project number AWCS#323-88-02-01. The NASA IV&V Facility, Code QV, provided the RTOP fund for the work. It was a joint effort between the Safety, Reliability, and Quality Assurance (SR&QA) Office and the Engineering Directorate of the Lyndon B. Johnson Space Center (JSC).

The technical manager of the project wishes to thank the following organizations and individuals for their technical contributions:

Dr. John C. Munson, QUEST Company, Moscow, Idaho; Professor of Computer Science Department, University of Idaho

Darrell S. Werries, QUEST Company, Moscow, Idaho

Dr. Norman F. Schneidewind, Professor of Computer Science & Director of Laboratories, Information Systems Group, Naval Postgraduate School

Ted Keller, Lockheed Martin Space Information Systems

Dr. Troy Henson, Lockheed Martin Space Information Systems

Trent Metcalf, Lockheed Martin Space Information Systems

Charlie Najvar, Lockheed Martin Space Information Systems

Patti Thornton, Lockheed Martin Space Information Systems

Malcolm J. Himel, Jr. Chief, Station Safety and Mission Assurance Branch, JSC SR&QA Office

Marcia C. Kerr, Flight Software Branch, Avionic System Division, JSC* Engineering Directorate

Alice T. Lee, Project Technical Manager
Assurance Analysis Branch
Flight Systems Safety and Mission Assurance Division
JSC SR&QA Office

This publication is available from the Center for AeroSpace Information, 800 Elkridge Landing Road, Linthicum Heights, MD 21090-2934 (301) 621-0390.

Contents

Contents

(continued)

Contents

(continued)

Figures

Contents

(concluded)

Page

Tables

Acronyms

BDF	Boolean discriminant function
CDF	cumulative distribution functions
DR	discrepancy reports
DRAT	Dynamic Reliability Assessment Tool
IV&V	independent verification and validation
LOC	lines of code
M.S.E.	mean square error
NASA	National Aeronautics and Space Administration
OPS	operations sequence
PASS	primary avionics software system
PCA	principal components analysis
RCM	relative complexity metric

Introduction

As software is maintained or reused, it undergoes an evolution which tends to increase the overall complexity of the code. To understand the effects of this, we brought in statistics experts and leading researchers in software complexity, reliability, and their interrelationships. These experts' project has resulted in our ability to statistically correlate specific code complexity attributes, in orthogonal domains, to errors found over time in the HAL/S flight software which flies in the Space Shuttle. Although only a prototype-tools experiment, the result of this research appears to be extendable to all other National Aeronautics and Space Administration (NASA) software, given appropriate data similar to that logged for the Shuttle onboard software.

Different flight scenarios cause the software system to execute different subsets of code of varying complexity. Some flight scenarios will execute functionally complex code sequences and will thus have a very strong likelihood of failure. Other flight scenarios will execute only code sequences of limited functional complexity and, thus, have higher reliability.

The reliability of a software system may best be modeled with those techniques that incorporate the functional relationship between software complexity attributes and software faults. Software models will demonstrate considerable variability on the complexity attributes. As the various models are executed in response to varying software inputs, the actual degree of exposure to fault-prone code will also vary. In essence, the reliability of a software system is not a static measure. It will be directly related to the varying functionality of the software.

In addition to observed failure history, reliability models should incorporate information as to the internal nature of the software that is actually executing. Earlier research has established that the functional complexity induced by each of the various functions that primary avionics software system (PASS) software may execute will be a direct determinant of this reliability. It is, therefore, the intent of this research to incorporate complexity effects into reliability modeling techniques.

The process of software reliability modeling using this enhanced approach is complicated by the fact that most software systems contain code modules of varying levels of maturity. This is the result of an effort to ensure maximum use of existing software by reusing it or by modifying code to control development costs. Software reliability measurement techniques should reflect the differences in reliability between older, more mature and newer, less-tested software modules in the same software system. To ensure measurement accuracy, the reused or modified code will be modeled differently than code whose reliability is yet to be determined.

The primary objective of this project was to formulate a process that can be readily applied to future software development projects, to be achieved through the modification of existing reliability models to

incorporate measures of dynamic program complexity. Existing models of software reliability were to be enhanced to incorporate two distinct aspects of variability previously unaccounted for in reliability models: reflect the internal structure of the software in the modeling process, and reflect the varying levels of software module maturity within a system.

The first project phase, in 1994, focused on data collection and management processes surrounding the software development process. The 1995 phase focused on the development of mathematical models of software reliability that reflect both software complexity and software maturity. The work was divided into seven milestones, all of which are discussed in this document in the order of sections. In Section I, we show guidelines developed for statistical testing of the Space Shuttle PASS software. New data derived will provide the basis for the following assessments:

- The adequacy of a software test
- The probability distribution for the execution profiles
- The testability of the PASS
- Design rules for future PASS software development

In Section II, we provide the following enhancements to mathematical software reliability prediction:

- Reliability modeling with functional complexity
- Parameter estimation
- Empirical validation
- User guidelines

In Section III, we provide an enhanced version (3.1) of the HAL/S Metric Analyzer (HALMet) which will now

- Execute on a number of additional workstation environments.
- Be integrated into the new Space Shuttle software development methodology being designed.

Deliverables also include the companion software tool, the principal components analysis-relative complexity metric data analysis tool called PCA-RCM.

The HALMet 3.1 tools package installation manual and the operation manual are provided in Appendixes A and B, respectively. The PCA-RCM 2.0 tool installation manual and operation manual are provided in Appendixes C and D, respectively.

In Section IV, we present a report that includes the following:

- Execution profile data for PASS running on a number of different operations sequences (OPS)
- An assessment of the variability of the functional complexity of the system
- An assessment of regression test efficiency, which allows a software tester to identify potential faults that were introduced in changes to a program

Section V provides a final review demonstration of the HALMet at the NASA Johnson Space Center (JSC).

Section VI describes how we classified severity levels of software faults and researched constructing a statistical software filter to help identify regions of the on-board flight software more prone to Severity 1 failures.

In Section VII, we describe software requirements specification and preliminary design documents developed for a reliability assessment toolset, and the software implemented and demonstrated at JSC.

Finally, we contracted with Dr. Norman F. Schneidewind to provide an enhancement of his Schneidewind software reliability model using metrics to improve prediction accuracy. His results can be found in Appendix E.

Our research has demonstrated that a more complete domain coverage can be mathematically demonstrated with the approach we have applied, thereby ensuring full insight into the cause-and-effects relationship between the complexity of a software system and the fault density of that system. By applying the operational profile we can characterize the dynamic effects of software path complexity under this same approach. We now have the ability to measure specific attributes which have been statistically demonstrated to correlate to increased error probability, and to know which actions to take, for each complexity domain. Shuttle software verifiers can now monitor the changes in the software complexity, assess the added or decreased risk of software faults in modified code, and determine necessary corrections.

The reports, tool documentation, user's guides, and new approach that have resulted from this research effort represent advances in the state of the art of software quality and reliability assurance. Details describing how to apply this technique to other NASA code are contained in this document.

Section I
Statistical Testing of the
Space Shuttle Primary Avionics Software System

Abstract

Traditional software testing practice assumes that faults in a computer program may be found by the application of rigorous deterministic processes. Computer software systems, however, are rapidly growing in both size and complexity. This paper presents a new view of statistical testing and provides for the redefinition of basic test objectives and for the optimal allocation of test resources. The term statistical testing will be used in conjunction with the statistical properties of the functional complexity of a software system and the number of faults it may contain. Testing effort should be concentrated on those functions that result in high functional complexity. From a statistical testing perspective, the adequacy of the test process is based on two distinct criteria: that sufficient testing has occurred across the range of functional complexity to provide reasonable estimates of the measures of central tendency and variability for functional complexity, and that adequate stress testing has occurred by identifying those functions that maximize its functional complexity.

Introduction

A significant problem with modern software testing procedures is that the objectives of the test process are not clearly specified and sometimes not clearly understood. If questioned, most software testers will associate the testing process with a process of finding faults in programs [Morell]. If this is the case, is the purpose of testing to find all of the faults or just some of the faults? How will we know when to stop testing? An implicit objective of the deterministic school of testing is to design some sort of a systematic and deterministic test procedure that will guarantee sufficient test exposure for the random faults distributed throughout a program [c.f., Frankl; Lou; Nakajo; Weyuker]. However, there is reason to believe that it is possible to identify measurable program attributes such as complexity that can explain a large proportion of the variance observed in the software faults in their location in software modules [Khoshgoftaar, 1992; Khoshgoftaar, 1990; Munson, 1990, "Regression…"]. This being the case, it would only seem reasonable to look for software faults in the complex code segments rather than introduce them only to see if we could locate these same injected faults again as is the case with previous approaches to statistical testing [Mills].

In keeping with the rapid increase in the functionality and capability of computer hardware, the average size of computer programs has grown consistently. In the 1960s, programs were measured in terms of hundreds of lines. In today's software development technology, programs are measured in terms of millions of lines. As programs have increased in length several orders of magnitude in the last three decades, the problems associated with testing these programs have also increased several orders of magnitude. The rapid growth in the size of new software systems has brought into question the viability of deterministic test

methodologies. Techniques need to be developed to serve as a filter for the code to identify regions of the code that are most likely to contain faults. We must also come to accept the fact that some faults will always be present in the code. The objective of the testing process is simply to find those faults that will have the greatest impact on the safety/survivability of the code. Under this new view of the software testing process, the act of testing may be thought of as conducting an experiment in the behavior of the code under typical execution conditions. We will determine, a priori, exactly what we wish to learn about the code in the test process and conduct the experiment until this stopping condition has been reached.

One of the fundamental tenets of the statistical approach to software test is that it is possible to create fault surrogates. While we cannot know the numbers and locations in faults, we can, over time, build models based on observed relationships between faults and some other measurable software attributes. Software faults and other measures of software quality can be known only at the point the software has finally been retired from service. Only then can it be said that all the relevant faults have been isolated and removed from the software system. On the other hand, software complexity can be measured very early in the software life cycle. In some cases, these measures of software complexity may be extracted from design documents. Some of these measures are very good leading indicators of potential software faults. The first step in the software testing process is to construct suitable surrogate measures for software faults. In the case of this study, a single measure, relative complexity, will be constructed to serve as such a surrogate. In those program modules that have large values of this surrogate measure, there is reasonable evidence to support the conclusion that the numbers of faults will also be large. If the code is made to execute a significant proportion of time in modules of large relative complexity, then the potential exposure to software faults will be great and, thus, the software will be likely to fail.

Measurement of Software Attributes

If we accept the premise that faults are introduced in code by systematic errors committed by programmers, then it should be possible to identify a set of measurable software attributes that are distinctly related to the conditions that lead to faults. Indeed, there are certain complexity metrics that have been shown to be distinctly associated with software faults [Munson, 1990, "Regression…"]. The methodology for identifying these attributes and their relationship to software faults has been demonstrated in several antecedent studies [Munson,1990, "Regression…"; 16]. The main point here is that, if we can identify those software attributes that are associated with faults, we may use these data to identify regions of code of software currently under development and test that are likely to contain faults.

There are two distinct and separable issues now. First, it is one thing to have a software system that contains faults. Second, it is quite another for the regions of code containing faults to be executed. If there does not exist a set of input data that will drive a program into a region of code that is likely to contain faults, then there will probably be few failures during the processing of this input set. If, on the other hand,

most every input data set will drive the code into the faulty code regions, then the software will prove quite failure prone.

If software measurement techniques are to be used in the testing process the next reasonable question is, what is the *best* single metric to use to represent the internal nature of a program? There are now over 100 such candidate metrics. Each, according to its author, clearly outperforms and eclipses all of the others. In truth, there is a high degree of interrelationship among all of the metrics. Recent research suggests that there are probably no more than four or five distinct complexity domains that are measured in some degree by each of the existing metrics [Munson, 1989]. If this is true, the best metric is, in fact, a set of metrics chosen to represent as much variance in the underlying complexity domains as is possible.

In most linear modeling applications concerned with the mapping of software metrics onto software faults, such as regression analysis and discriminant analysis, the independent variables or metrics are each assumed to represent some distinct aspect of variability not clearly present in other measures. In software development applications, the independent variables—in this case, the complexity metrics—are well correlated and thus demonstrate a high degree of multicollinearity. Such models may be subject to dramatic changes due to additions or deletions of variables or even discrete changes in metric values. To circumvent this problem, principal components analysis has been used, quite successfully, to map the metrics into orthogonal attribute domains [Munson, 1989]. Each principal component extracted by this procedure may be seen to represent an underlying common attribute domain.

The relative complexity, ρ, of the factored program modules may be represented as follows:

$$\rho_i = \sum_j \lambda_j d_{ji}$$

where λ_j is the eigenvalue associated with the j^{th} factor and d_{ji} is the j^{th} domain metric of the i^{th} program module on the j^{th} domain. Each of the eigenvalues represents the relative contribution of its associated domain to the total variance explained by all of the domains. In essence, then, the relative complexity metric is a weighted sum of the individual domain metrics. In this context, the relative complexity metric represents each raw complexity metric in proportion to the amount of unique variation contributed by that complexity metric.

Program Functions and Operations

Software systems are designed to implement each of their functionalities in one or more code modules. In some cases there is a direct correspondence between a particular program module and a particular functionality. That is, if the program is expressing that functionality, it will execute exclusively in the module in question. In most cases, however, there will not be this distinct traceability of functionality to modules. The functionality will be expressed in many different source code modules.

Assume that the software system *S* was designed to implement a specific set of mutually exclusive functionalities *F*. Thus, if the system is executing a function $f \in F$ then it cannot be expressing elements of any other functionality in F. Each of these functions in F was designed to implement a set of software specifications based on a user's requirements. From a user's perspective, this software system will implement a specific set of operations, *O*. This mapping from the set of user perceived operations, *o*, to a set of specific program functionalities is one of the major functions in the software design process. It is possible, then, to define a relation IMPLEMENTS over $O \times F$ such that IMPLEMENTS(o, f) is true if functionality *f* is used in the design process to implement an operation, *o*. For each operation $o \in O$, there is a relation p' over $O \times F$ such that $p'(o, f)$ is the proportion of time assigned to functionality *f* by operation *o*.

A typical non-functional requirement for modern systems (particularly safety critical systems) is that these systems perform reliably. One method of ensuring that this non-functional requirement will be met is to subject the software system to a series of tests. Each test will exercise one or more of the functionalities in *F*. Each possible test of the software system will select a subset of *F*. There is a relation EXHIBITS over $T \times F$ such that EXHIBITS(t, f) is true if test case *t* exhibits functionality *f*.

From a software design standpoint a system is generally decomposed into many program modules. For a given software system, *S*, let *M* denote the set of all program modules for that system. Let *T* denote the set of all test cases in a test suite for *S*. For each test case $t \in T$, there is a relation p'' over $T \times M$ such that $p''(t,m)$ is the proportion of time assigned to module *m* by test *t*. Let us assume that there is a relationship between program functionalities and software modules. Program modules may be viewed in terms of a number of different subsets. Some modules may execute under all of the functionalities of *S*. These are the set of common modules. The main program is an example of such a module that is common to all tests of the software system. These common modules, $M_c \subset M$ are defined as

$$M_c = \{m: M | \forall t \in T \bullet p''(t, m) > 0\}$$

All of these modules will execute regardless of the specific functionality being executed by the software system.

Yet another set of software modules may or may not execute when the system is running a particular function. These modules are said to be potentially involved modules. The set of potentially involved modules is

$$M_p^{(f)} = \{m: M | \exists t \in T \bullet \text{EXHIBITS}(t, f) \wedge p''(t,m) > 0\}$$

In other program modules, there is extremely tight binding between the functionality and the module. That is, every time a particular function is executed, a distinct set of software modules will always be invoked. The set of indispensably involved modules is

$$M_i^{(f)} = \{m{:}M \mid \forall t \in T \bullet \text{EXHIBITS}(t, f) \Rightarrow p''(t, m) > 0\}$$

For the execution of a function, *f*, the set of modules, M_f that are designed to implement this function is

$$M_f = M_c \cup M_p^{(f)} \cup M_i^{(i)}$$

Clearly, for the test process to be effective, the range of functionalities, *f*, must be well known.

Within a test suite, each test will exercise one or more of the system's functionalities. For a given test *t* these expressed functionalities are those with the property

$$F_e^{(t)} = \{f{:}F \mid \forall t \in T \bullet \text{EXHIBITS}(t, f)\}$$

For any test *t* the execution profile will consist of the probabilities $p''(t,m)$ derived from

$$m \in M_t = M_c \cup M_p' \cup M_i(F_e^{(t)})$$

where $M_p' = \bigcup_{F_e^{(t)}} M_p(F_e^{(t)})$, and $M_i(F_e^{(t)})$ are the modules that are actually invoked in the test *t*.

From the standpoint of software design and the test process, the real problems introduced into testing are not necessarily attributable to the set of modules that are tightly bound to a functionality M_i or to the set of common modules that will be invoked for all executing processes M_c. The real problem is the set of potentially invoked modules M_i. The greater the cardinality of this set of modules in relation to M_i the less certain will be the outcome of a test of this functionality. For any one test of this functionality, none of the modules may execute or they may all execute.

The Estimation of Profiles

Each test suite will implement a subset of functionalities, i.e., $F_e^{(t)} \subset F$. As each test is run to completion it will generate a test *execution profile*. This test execution profile may represent the results of the execution of one or more functions. To simplify the discussion let us assume that a test expresses precisely one functionality. When a program begins the execution of a functionality we may envision this beginning as the start of a stochastic, Markov process. For the system *S* there is a call graph that shows the transition of program control from one program module to another. The transition from one module to another may be seen as a stochastic process. In which case we may define an indexed collection of random variables $\{X_t\}$, where the index *t* runs through a set of non-negative integers, $t = 0, 1, 2, \cdots$ representing the epochs of the process. At any particular epoch the software is found to be executing exactly one of its *M* modules. The fact of the execution occurring in a particular module is a *state* of the system. For this software system, the system is found in exactly one of a finite number of mutually exclusive and exhaustive states

that may be labeled $1, 2, \cdots, M$. In this representation of the system, there is a stochastic process $\{X_t\}$, where the random variables are observed at epochs $t = 0, 1, 2, \cdots$ and where each random variable may take on any one of the M integers, from the stated space $A = \{1, 2, \cdots, M\}$.

The probability that a particular module may execute is, in fact, a conditional probability. Let *Y* be a random variable defined on the indices of the set of elements of *F*. Then $p_i^{(k)} = \Pr[X_n = i | Y = k]$ where $k = 1, 2, \cdots, \#\{F\}$ represents the execution profile for a set of modules executing a function *k* exclusively. The distribution of the execution profile is also multinomial for a software system consisting of more than two modules.

As a matter of the design of a program, there may be a non-empty set $M_p^{(f)}$ of modules that may or may not be executed when a particular functionality is exercised. This will, of course, cause the cardinality of the set M_f to vary. A particular execution may not invoke any of the modules of $M_p^{(f)}$. On the other hand, all of the modules may participate in the execution of that functionality. As we will see, this variation in the cardinality of M_f within the execution of a single functionality will contribute greatly to the amount of test effort that will be necessary to test such a functionality.

Functional Complexity

The execution profile for a program can be expected to change across the set of program functionalities. This will result in a concomitant variation in the *functional complexity* of a program [Munson, 1992]. In other words, for each functionality, f_i, there is an execution profile represented by the probabilities $p_1^{(i)}, p_2^{(i)}, p_3^{(i)},, \cdots, p_n^{(i)},$. As a consequence, there will be a functional complexity $\phi_p^{(i)}$ for the execution of each function, f_i , where

$$\phi_p^{(i)} = \sum_{j=1}^{n} p_j^{(i)} \rho_j .$$

This is distinctly the case during the test phase when the program is subjected to numerous test suites to exercise differing aspects of its functionality. The functional complexity of a system will vary greatly as a result of the execution of these different test suites.

It is possible to determine the functional complexity for various functionalities at varying stages of software maturity and thus to understand their likely reliability. To test effectively a software system we must necessarily determine the functionality of the program and how these functions will interact as the program executes. The latter information not only directs the formation of test suites but also provides the information necessary to formulate execution profiles. The functionalities that imply execution profiles which cause the functional complexity to increase merit our attention since these are the conditions that will increase failure rates for a given design.

Statistical Testing

The central thesis of statistical testing is that programs will fail because of their indigenous faults that are in turn directly related to measurable software attributes. If you can understand the relationship between faults and specific attributes, you will know where to look for faults. Of particular utility, in this role as a fault surrogate, is relative complexity as a static measure of program complexity and functional complexity as a dynamic measure of program complexity. The important thing for the following discussion is that there exists some measurable program module attribute, say ϕ that is highly correlated with software faults. Intuitively (and empirically), a program that spends a high proportion of its time executing a module set M_f of high ϕ values will be more failure prone than one that seeks out program modules of smaller values of this attribute.

From the discussion earlier, we see that a program may execute any one of a number of basic functionalities. Associated with each of the functionalities there is an attendant value of ϕ. At the beginning of the test process, the first step in statistical testing is to understand the nature of the variability of ϕ ***within*** the functions that the program may execute. The next step in the statistical testing process is to understand the nature of the variance of ϕ ***between*** functionalities. The greater the variability in the ϕ of a program found during normal test scenarios, the more testing will be required. It is normally presumed that a major objective of software testing is to find all of the potential faults in the software. We cannot presume to believe that a test has been adequate until we have some reasonable assessment of the variability in the behavior of the system.

In this new perspective, the whole nature of the test process is changed. Our objective in this new approach will be to understand the program that has been designed. It may well be that the design of a program will not lend itself to testing in any reasonable time frame given a high ratio of *within* module variability to *between* module variability. We cannot hope to determine the circumstances under which a program might fail until we first understand the behavior of the program itself.

The next logical step in the statistical testing process is to seek to identify the precise nature of the mapping of functionality to ϕ. That is, we need to identify the characteristics of test scenarios that cause our criterion measure of ϕ to be large. Test scenarios whose values of ϕ are large are those that will most likely provide maximum exposure to the latent faults in a program. In this new view, a program may be ***stress tested*** by choosing test cases that maximize ϕ.

The initial stage in the testing process is concerned primarily with developing an understanding of the behavior of the software system that has been created by the design process. As such, all test suites during this phase will be carefully architected to express a single functionality. For each of these program functionalities f_i let us define a random variable $\boldsymbol{a}^{(i)}$ defined on the domain of values of the functional complexity $\phi_f^{(i)}$ of the i^{th} function. As each test of functionality f_i is conducted, we will have a sample

data point $a_j^{(i)}$ on $\boldsymbol{a}^{(i)}$. After a sequence of tests of f_i we may compute a sample mean $\bar{a}^{(i)} = \sum_j a_j^{(i)}$, an estimate of the parameter $\mu^{(i)}$, the mean functional complexity of the i^{th} functionality. Similarly it will be possible to compute the sample variance $s_{(i)}^2 = \frac{1}{n}\sum_j (a_j^{(i)} - \bar{a}^{(i)})^2$, an estimate of the parameter, $\sigma_{(i)}^2$, the variance of $\phi_f^{(i)}$ for the i^{th} functionality. Note that the contribution in variability of each test is strongly influenced by the set $M_p^{(f)}$. If this is an empty set, the range of the functional complexity for a given functionality will be considerably constrained.

Without any loss in generality, let us assume that the $\boldsymbol{a}^{(i)}$ are defined on a normal probability distribution. This distribution is specified succinctly by its mean $\mu^{(i)}$ and its variance $\sigma_{(i)}^2$ of which the sample mean $\bar{a}^{(i)}$ and variance $s_{(i)}^2$ are sufficient statistics. It is now possible to construct a standard error of the estimate $\bar{a}^{(i)}$ as follows: $s_{\bar{a}^{(i)}} = \frac{s_{(i)}}{\sqrt{n}}$ for a set of *n* tests of the i^{th} function.

Once we have initiated a testing phase it would be desirable to be able to formulate a mathematical statement or a stopping rule for determining the conclusion of the test phase. A stopping rule for this first test phase is based on the attainment of an a priori condition for *b* and α that

$$\Pr\left[-b < \frac{\bar{a}^{(i)} - \mu^{(i)}}{\sigma_{(i)}/\sqrt{n}} < b\right] = 1 - \alpha .$$

In other words, the initial test phase will continue until we have attained an estimate for each of the mean functional complexities of the software system with the $100(1-\alpha)\%$ confidence limits set as a condition of a test plan.

As the test process progresses during this first test phase, it may well emerge that the variation in the functional complexity may be inordinately large. We can begin to see in advance that a significant amount of total test resources will be consumed only on this first parametric estimation phase of the software. In this circumstance it is quite possible that the software system is not testable. This being the case, we are now in a position to specify that the implementation of certain functionalities, specifically those for which the *within* functionality variability is so large as to demand too many test resources, be redesigned and recoded. Not all software may be reasonably or economically tested. This view represents a fundamental shift in test philosophy. In the past, software testers were obliged to take their best shot at whatever systems were designed and coded by a staff outside the test process. Through the systematic introduction of measurement processes in the testing phase of the life cycle, it is now possible to set criteria a priori for the testability of software systems. In looking at the parameter estimation phase of software testing, we are working strictly with the *within* functionality variability of a design. For the next phase of the software test process, let us focus on the variability among the set of all functionalities. The focus in this next phase is

on the construction of a sequence of tests that will focus on the set of operations, *O*, that will represent the user's view of the system.

It would be entirely unrealistic to suppose that it would be possible to construct a fault-free software system or to test a large system exhaustively to find all possible faults in the system. It is not really of interest that a system has faults in it. What is relevant, however, is that there are no faults in the range of functions that the consumer of the software will typically execute. It would take unlimited test resources to certify that a system is, in fact, fault free. What we would like to do is to allocate our finite test resources to maximize the exposure of the software system to such faults as might be present.

Let us assume that we are investigating a system designed to implement a set of *n* functional requirements. As a result of the design process, each of these functions f_i will have an associated functional complexity $\phi_f{}^{(i)}$. From the initial stages of the test process we will have an estimate a_i for $\phi_f{}^{(i)}$. We can now formulate an objective function for the allocation of test resources as follows:

$$Q = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n$$

where x_i is the amount of test resources (time) that will be allocated to the test of the i^{th} function and *Q* is a measure proportional to software faults (quality). It would be well to remember that functional complexity is an expected value for the relative complexity of the design implementation of each functionality. Relative complexity was constructed to be a surrogate for software faults. Thus, a functionality whose functional complexity is large may be expected to be fault prone. By maximizing the value of *Q* in the objective function above, a test plan will be seen to have maximized its exposure to the embedded faults in the system. Clearly the best way to maximize *Q* is to allocate all test resources to the function f_i whose functional complexity is the largest. That is, allocate all resources to a_i where $a_i > a_j$ for all $j \neq i$.

The real object of the test process, at this stage, is to maximize our exposure to software faults that will have the greatest impact on those functions that the user will be performing with the software. In this sense, the test process is constrained by the customer's operational profile. As per the earlier discussion, a user of the software has a distinct set of operations, *O*, that he/she perceives the software will perform. These operations are then implemented in a rather more precise set of software functions, *F*. From this operational profile we may then construct the following constraints on the test process to ensure that each of the user's operations receives its due during the test process.

$$\begin{gathered} b_{11} x_1 + b_{12} x_2 + \cdots + b_{1n} x_n \leq c_1 \\ b_{21} x_1 + b_{22} x_2 + \cdots + b_{2n} x_n \leq c_2 \\ \vdots \\ b_{m1} x_1 + b_{m2} x_2 + \cdots + b_{mn} x_n \leq c_m \end{gathered}$$

The coefficients b_{ij}, are proportionality constants that reflect the implementation of an operation in a set of appropriate functions such that:

$$b_{ij} = \begin{cases} p'(o_i, f_j) & \text{if IMPLEMENTS}(o_i, f_j) \text{ is } \textit{true} \\ 0 & \text{if IMPLEMENTS}(o_i, f_j) \text{ is } \textit{false} \end{cases}$$

and c_i represents the test resources assigned to the i^{th} operation. The total test effort is simply the sum of time apportioned to each of the functionalities,

$$\sum_{i=1}^{n} x_i = \sum_{j=1}^{m} c_j \, .$$

Test Results

To demonstrate the concept of measurement of the testing process, the test results for a sequence of tests of a past release of the space shuttle PASS will now be examined. Table 1 summarizes these test results. To initiate this process, we used the metric tool, HALMet, to obtain the 19 complexity metrics for each of 572 higher-level program modules for PASS. We then transformed these 19 measures on 572 modules into 4 orthogonal measures as per the earlier discussion of principal components analysis. Finally, we then developed relative complexity measures for each of the 572 program modules. These represented the static measure of program complexity for each program module. It should be noted that the values actually computed for relative complexity were adjusted so that they had a mean of 50 and a standard deviation of 10. Thus, a program module that had a relative complexity of 55 would be one whose complexity was approximately one-half a standard deviation above the mean for all of the 572 program modules.

As each test was run in each test suite, we obtained the execution profiles for the 572 program modules. These values, again, represent the proportion of time that was spent in each of the program modules for each of the tests. We then computed the functional complexity of each program module. The system functional complexity under each of the tests was the sum of all of the functional complexities for all of the program modules (most of which are zero in that each test would only exercise a limited number of program modules). A test of average complexity is one with a functional complexity of 50.

Table 1 presents the results of the execution of four distinct test suites. The test suites are represented by columns in this table. Each test suite represents the exercise of a distinct functionality of the PASS. Within each test suite there are a varying number of tests. For example, in the case of G1 Ascent, there are a total of three tests. For each of the test suites, the *within* test statistics are then shown in this table. These test statistics include the mean functional complexity $\bar{a}_i^{(j)}$ for each test suite, the *within* functionality variance $s_{(i)}^2$, the standard error of the estimate for the test suite mean $s_{\bar{a}^{(i)}}$, and the 95% confidence

intervals for the test suite mean. Table 1 also shows the cardinalities for the sets M_c, M_i, and M_p, for each of the test functionalities.

The test suites are most unusual in that there is extremely little variation in their functional complexity both within the test suites and across all test suites. The G3 Entry tests are the least complex of the series. Both the G1 Ascent test and the G1/G6 RTLS tests represent the most complex in the series. The test suites also clearly differ in terms of their variability. Though the total number of test observations is small, we can see that there is more variability within the G3 tests than within the G1 tests. The G3 test suites are significantly less functionally complex that the G1 series. In that the level of functionality is of high granularity, the functional complexity of the entire test is very close to the expected value of 50. The real value of the statistical testing process comes with a more fine-grained test scenaio. The current test data, though, do demonstrate the viability of the approach.

Table 1. Test Results

Test Suite	G1 Ascent	G1/G6 RTLS	G3 Deorbit	G3 Entry
test				
1	52.53	52.79	51.41	48.06
2	53.02	52.17	49.31	49.31
3	52.80	52.17	50.09	49.29
4		52.18	50.60	49.30
$\bar{a}_i^{(j)}$	52.78	52.33	50.35	48.99
$s^2_{(i)}$	0.06	0.10	0.78	0.38
$s_{\bar{a}^{(i)}}$	0.14	0.16	0.44	0.31
lower 95%	52.50	52.02	49.49	48.38
upper 95%	53.07	52.63	51.22	49.59
M_c	92	92	92	92
M_i	80	150	98	105
M_p	3	7	11	10

It is also worth noting that the ratio of the cardinality of M_p, the potentially invoked modules, to M_i, the modules that are always invoked for the execution of a functionality is greatest with the G3 Deorbit test

suite. As may be seen in this table, the variance in functional complexity is also greatest for this test suite. Within the limited variance component of each test, this variance changes in proportion to the ratio of M_p to M_i.

Each of the columns in Table 1 represents the ***within*** function sources of variation in implementing each function. If we now compute the average of the average functional complexity ***between*** systems, we will have a *between* suite functional complexity of 51.00, just slightly above average relative complexity for the whole software system. The *between* suite variability is 2.74, indicating a remarkable lack of variation in functional complexity across the several functionalities.

Table 2 shows the average relative complexity of just those modules that actually executed in each of the test cases. Again, it is most unusual for the relative complexity of these module subsets to have values of average relative complexity so close to the PASS system average complexity of 50.00. This means that the static complexity of each of the module subsets was almost exactly the same as the complexity for the entire system. This is certainly not a chance nor a random phenomenom.

Table 2. Test Relative Complexity

Test Suite	G1 Ascent	G1/G6 RTLS	G3 Deorbit	G3 Entry
test				
1	51.08	50.52	50.74	50.25
2	50.70	50.51	50.74	50.34
3	52.08	50.51	50.74	50.38
4		50.43	50.67	50.38

Figures 1-6 depict a series of six frequency plots. Each of these frequency plots shows the distribution of the relative complexity of the modules in each of the test suites together with the combined plot of all 338 modules that were invoked in at least one of the four tests. A visual inspection of these charts shows a rather different distribution of relative complexity for each of the tests. In particular, the distribution of the module relative complexity for the G1 Ascent series is visually different from those in the other test series. In the presence of this apparent visual difference in distribution, it is even more remarkable that the average module relative complexity is practically invariant of the four test series as may be seen in Table 2 above.

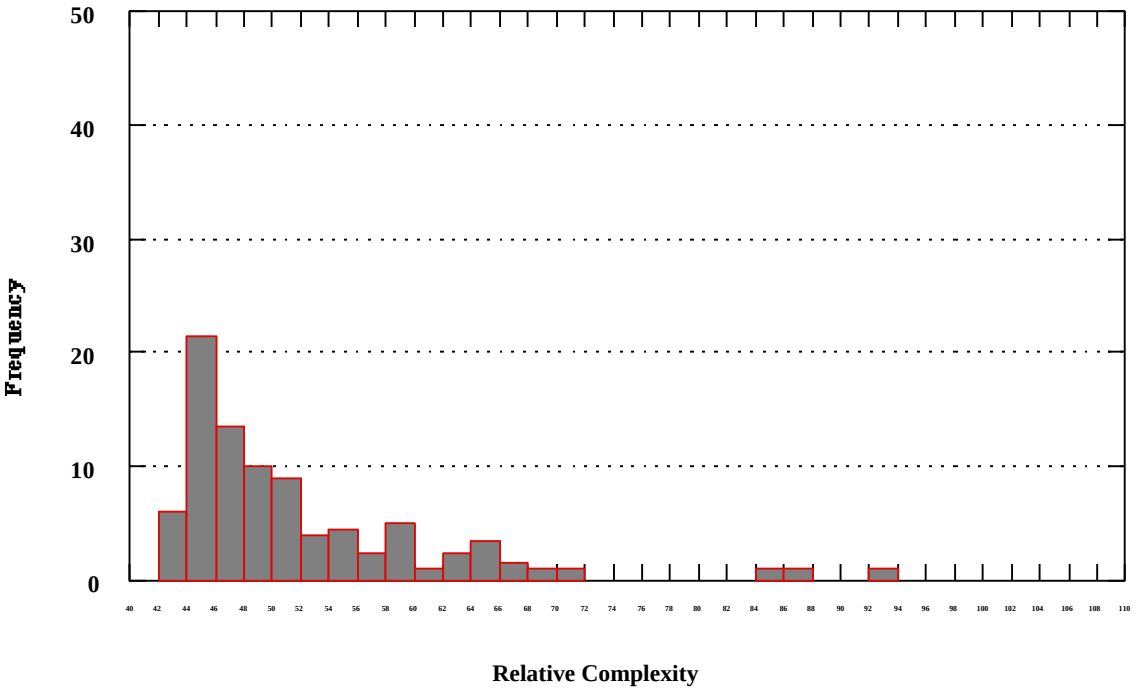


Figure 1. Module complexity for G1 ascent.

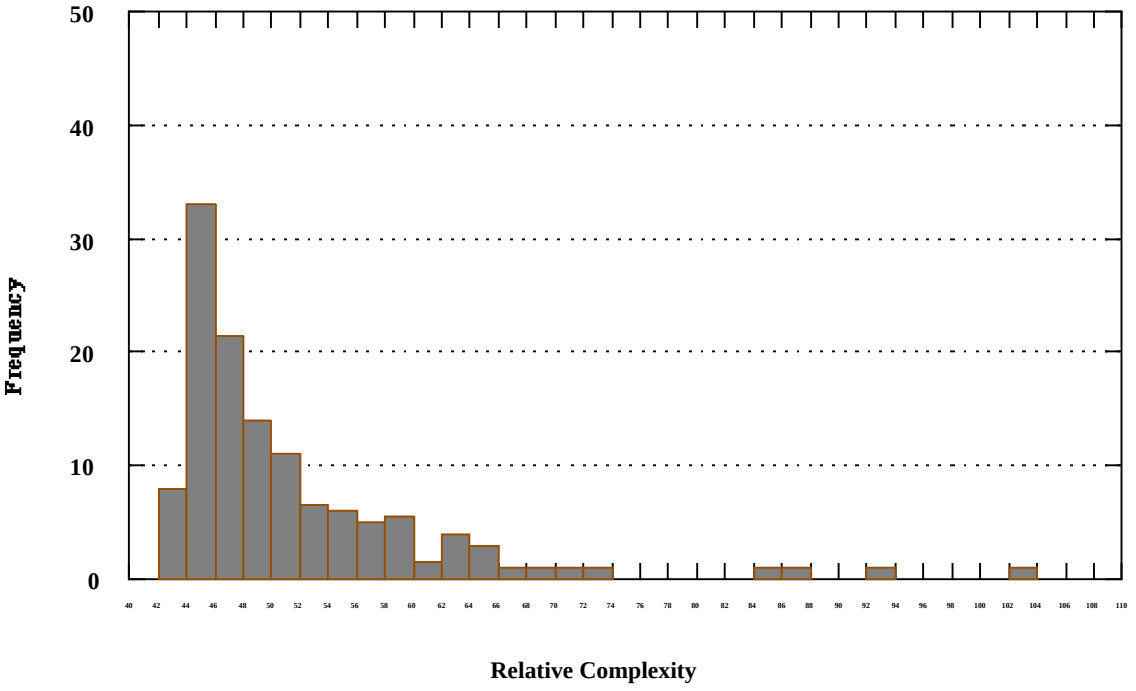


Figure 2. Module complexity for G1-G3 RTLS

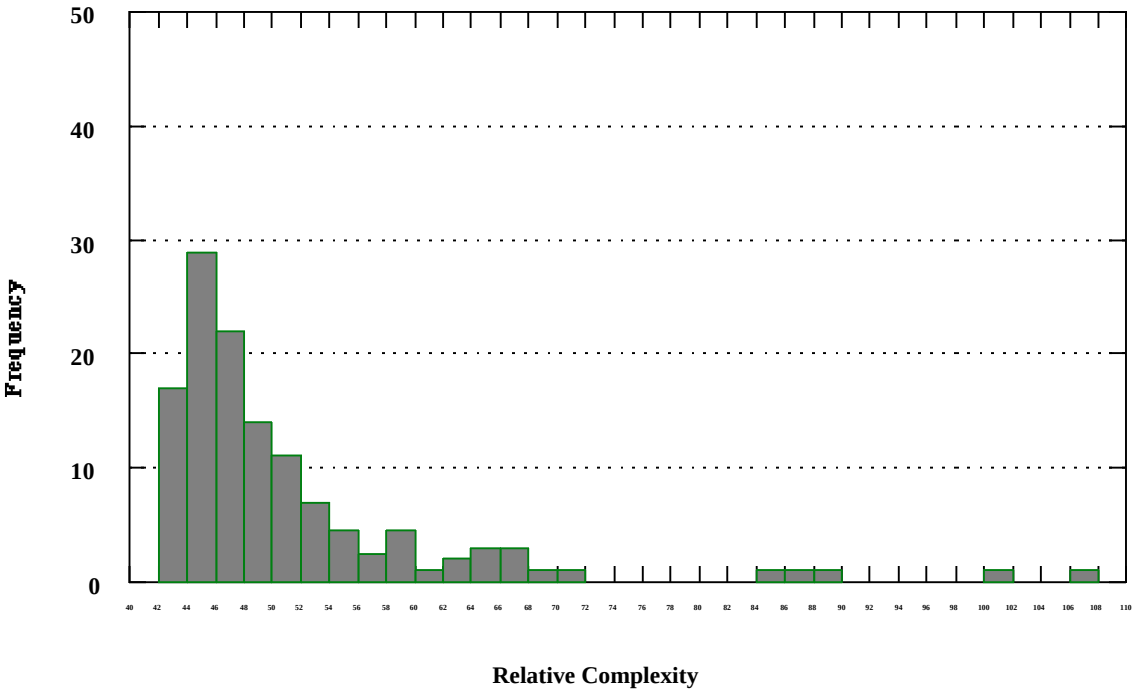


Figure 3. Module complexity for G3 deorbit.

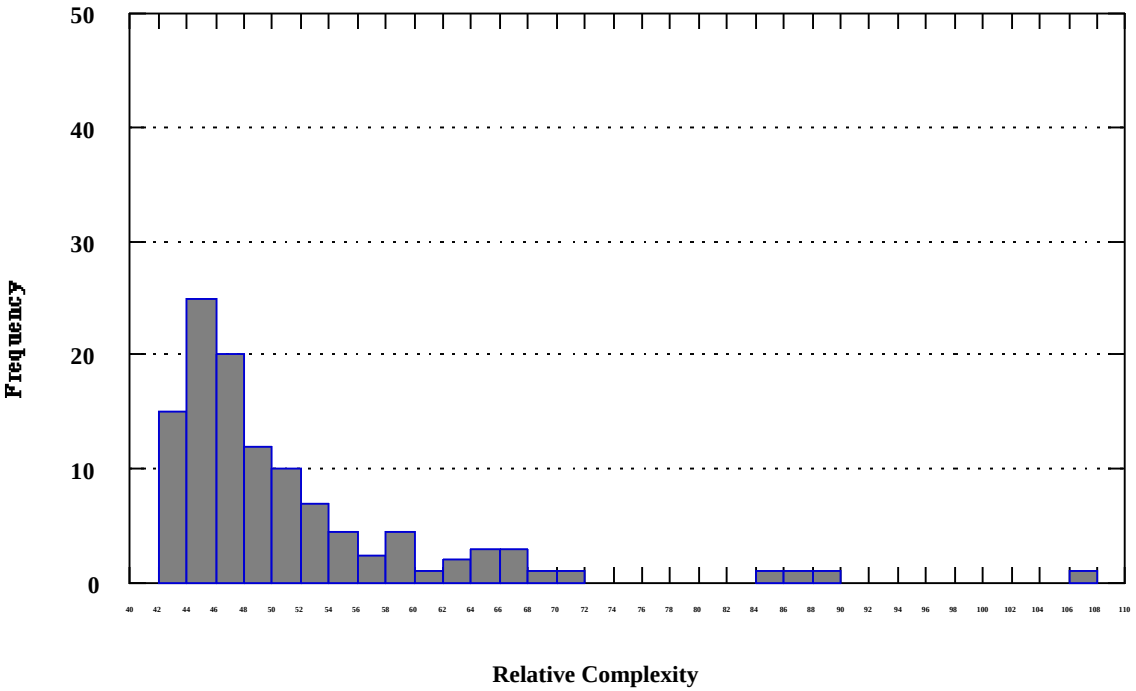


Figure 4. Module complexity for G3 entry.

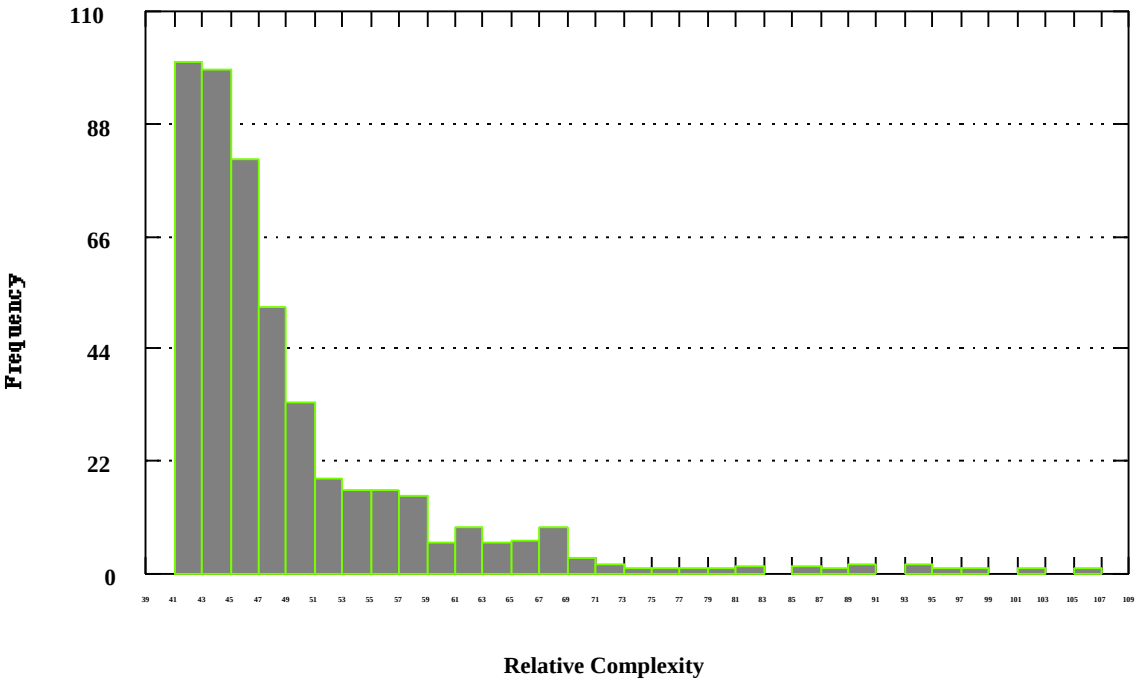


Figure 5. Module complexity for entire system.

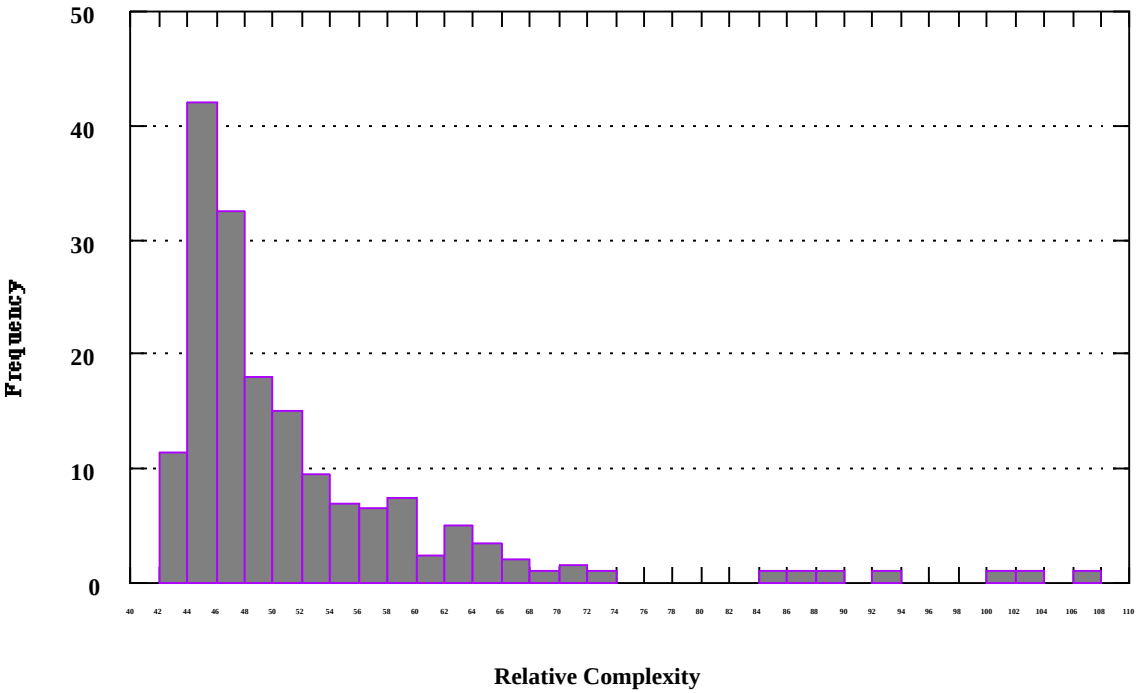


Figure 6. Module complexity for entire suite.

Summary and Conclusions

Complexity metrics can provide substantial information on the distinguishing differences among the modules of a software system regarding the conduct of the testing process. There is reasonable evidence to support the conclusion that computer software metrics may be used as leading indicators of software quality in the test process. Further, these metrics can be used (1) to guide the formulation of a test plan for structuring the test process and (2) to allocate test resources to yield a maximum exposure to software faults during the test process. An ancillary benefit is that we may now begin to consider some software configurations (designs) as being untestable within the limited time frame available for the whole testing process [Munson, 1993].

However high the level of test granularity might be for these tests, it is ***most*** unusual that 1) the functional complexity of each of the individual tests is so very close to the expected value of 50 and 2) there is so very little variability in the functional complexity. The only reasonable explanation for both points 1 and 2 is that this software system represents a very mature state of software development and that any undue variation in functional complexity would have been removed over time because of the inability to test the software effectively. This process is analogous to a rock newly cleaved from a steep cliff falling into a swiftly flowing river. The sharp edges on the rock will very rapidly be removed by the abrasive action of the stream yielding a rock with very little surface variation.

A similar observation could be made about the static complexity of the modules actually involved in each of the tests. The fact that there was so little variation in these static complexity measures is very surprising. This is also not a chance phenomenon. It must also be an artifact of the maturity of the PASS software.

The success of this technique is predicated on the ability to identify a surrogate for software faults. Relative complexity is a stand-in for aspects of software quality that we cannot directly measure such as software faults. From the results of our recent research, we believe that the relative complexity measure and functional complexity are stable and reasonable measures to use in the testing process. Unlike other metrics, the relative complexity metric combines, simultaneously, all attribute dimensions of all complexity metrics. We have established that software complexity metrics, and subsequently the relative complexity metric, are closely associated with measures of program quality and that this relationship may be exploited in the statistical testing process.

Section II, Results of Reliability Model Application

Abstract

Ongoing investigations into the etiology of software failures in the Space Shuttle PASS have provided substantial insight into the measurement of the reliability of this system. This review has led to the conclusion that it is not the software system that fails: It is the software system executing a particular functionality that fails. From this new perspective, the sequential execution of program functions may be modeled as a stochastic process. In particular, the program functionalities are physically expressed within a program as subtrees of modules in a program call tree hierarchy. The transitions between the program modules in a pairwise fashion may be represented in a transition matrix of a Markov process. Program failures are represented by an absorbing state in the transition matrix. This view of reliability permits the dynamic estimation of the parameters of the underlying multinomial probability distribution representing the transition between program modules. This use of the multinomial probability distribution is particularly useful in that it has a Dirichlet distribution as its natural conjugate prior distribution. Thus, a Bayesian approach may be employed so that each step or epoch in the software test process provides incremental information as to the evolving reliability assessment of the program.

This report represents the results of the second project milestone for this fiscal year. It formulates a new measure of the determination of the functional reliability and the overall reliability of the Space Shuttle PASS using a Baysian decision process based on a Markovian model of the system operation.

Introduction

The traditional concept of time as it is commonly used in software reliability modeling has little or nothing to do with the failure of computer software. Program failure is simply not time-dependent. A program may be viewed as a set of program modules that are executing a set of mutually exclusive functions. If the program executes a functionality consisting of a subset of modules that are error free, it will never fail, no matter how long it executes this functionality. If, on the other hand, the program is executing a functionality that contains fault-laden modules, there is a very good likelihood that it will fail. Further, it will fail with certainty when the right aspects of functionality are expressed. Another significant problem in the determination of the reliability of a software system is the precise determination of the failure event. The failure event, and the circumstances that surround the failure, have proven to be most elusive concepts. This section will explore an alternative view of software reliability, together with a mechanism for the collection of the data necessary to understand the failure.

The main problem in understanding software reliability is getting the granularity of the observation right. Software systems are designed to implement each of their functionalities in one or more code modules. In some cases there is a direct correspondence between a particular program module and a particular functionality. That is, if the program is expressing that functionality, it will execute exclusively in the module in question. In most cases, however, there will not be this distinct traceability of functionality to

modules. The functionality will be expressed in many different code modules. It is the individual code module that fails. A code module will, of course, be executing a particular functionality when it fails. We must come to understand that it is the functionality that fails.

As a program is exercising any one of its many functionalities in the normal course of operation of the program, it will apportion its time across this set of functionalities. The proportion of time that a program spends in each of its functionalities is the *functional profile* of the program. Further, within the functionality, it will apportion its time across one to many program modules. This temporal distribution of processing time is represented by the concept of the execution profile. In other words, if we have a program structured into *n* distinct modules, the *execution profile* for a given functionality will be the proportion of time spent in each program module during the time that the function was being expressed.

As the discussion herein unfolds, we will see that the key to understanding program failure events is the direct association of these failures to execution events with a given functionality. A Markovian stochastic process will be used to describe the transition of program modules from one to another as a program expresses a functionality. From these observations, it will become fairly obvious just what data will be needed to describe accurately the reliability of the system. In essence, the system will essentially be able to appraise us of its own health. The reliability modeling process is no longer something that will be performed *ex post facto*. It may be done dynamically while the program is executing, if need be. Since the reliability of a software system is a major concern specifically during software test activity, we will focus on the reliability assessment process during the test activity.

A Formal Description of Program Operation

Assume that the software system *S* was designed to implement a specific set of mutually exclusive functionalities *F*. Thus, if the system is executing a function $f \in F$ then it cannot be expressing elements of any other functionality in F. Each of these functions in *F* was designed to implement a set of software specifications based on a user's requirements. From a user's perspective, this software system will implement a specific set of operations, *O*. This mapping from the set of user-perceived operations, *o*, to a set of specific program functionalities is one of the major functions in the software design process. It is possible, then, to define a relation IMPLEMENTS over $O \times F$ such that IMPLEMENTS(*o*,*f*) is true if functionality *f* is used in the design process to implement an operation, *o*. For each operation $o \in O$, there is a relation p' over $O \times F$ such that $p'(o, f)$ is the proportion of time assigned to functionality *f* by operation *o*.

The main purpose, then, of the software specification process is to define the mapping of user-defined operations into functions that will be implemented by the software design process. An example of such a mapping may be found in Table 3, below. From this table it can be seen that a user's perceived operation o_1 has been implemented in two specific functions, f_1 and f_2.

Table 3. Mapping From Operations to Functions

$O \times F$	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
o_1	**T**	**T**							
o_2			**T**				**T**		
o_3			**T**			**T**	**T**		
o_4				**T**			**T**		**T**
o_5					**T**			**T**	**T**
o_6					**T**			**T**	

The software design process is strictly a matter of assigning functionalities in *F* to specific program modules $m \in M$, the set of program modules. The design process may be thought of as the process of defining a set of relations, ASSIGNS, over $F \times M$ such that ASSIGNS(*f*,*m*) is true if functionality *f* is expressed in module *m*.

The principal function of the software design process is to achieve the mapping from a set of functions defined in the specification process to a set of program modules that will express this functionality. An example of such a mapping may be seen in Table 4. From this table we can see that module m_1 will be executed in all of the specified functions. On the other hand, module m_2 is unique to function f_1. Other program modules, such as m_4 are used in two distinct functions.

A typical non-functional requirement for modern systems (particularly safety-critical systems) is that these systems perform reliably. One method of ensuring that this non-functional requirement will be met is to subject the software system to a series of tests. Each test will exercise one or more of the functionalities in *F*. Each possible test of the software system will select a subset of *F*. There is a relation EXHIBITS over $T \times F$ such that EXHIBITS (*t*,*f*) is true if test case *t* exhibits functionality *f*.

Table 4. Mapping From Functionality to Program Modules

$F \times M$	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
f_1	**T**	**T**								
f_2	**T**			**T**	**T**					
f_3	**T**			**T**		**T**	**T**			**T**
f_4	**T**				**T**					**T**
f_5	**T**									
f_6	**T**									
f_7	**T**		**T**				**T**			
f_8	**T**		**T**					**T**		
f_9	**T**		**T**						**T**	**T**

A designer of software test is charged with the responsibility of ensuring that the functionalities that will express the user's operations actually perform reliably after the tests have been conducted. The tester will architect a sequence of tests that will map specific test activities to specific functionalities. An example of this mapping may be seen in Table 5. From this table, we can see that test t_1 will express the functions f_1, f_4, and f_8.

Table 5. Mapping From Tests to Functions

$T \times F$	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
t_1	**T**			**T**				**T**	
t_2		**T**					**T**		
t_3			**T**			**T**			
t_4	**T**			**T**			**T**		**T**
t_5					**T**			**T**	

From a software design standpoint, a system may be decomposed into many program modules. For a given software system, *S*, let *M* denote the set of all program modules for that system. Let *T* denote the set of all test cases in a test suite for *S*. For each test case $t \in T$, there is a relation *p* over $T \times M$ such that $p''(t,m)$ is the activation frequency of module *m* in test *t*. In a continuation of the hypothetical example of

test mapping from Table 5, listed below in Table 6 are the proportion of module executions for each of the modules exercised in each of the tests. From this table we can see, for example, that under test t_1 module m_4 has received control in at least half of the module transitions during the execution of this test. Modules m_5, m_6 and m_9 have not executed at all during the test.

Table 6. Mapping From Tests to Program Modules

$T \times M$	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
t_1	**T**	**T**								
t_2	**T**			**T**	**T**					
t_3	**T**			**T**		**T**	**T**			**T**
t_4	**T**				**T**					**T**
t_5	**T**									

Let us assume that there is a relationship between program functionalities and software modules. Program modules may be viewed in terms of a number of different subsets. Some modules may execute under all of the functionalities of *S*. There will be the set of common modules. The main program is an example of such a module that is common to all tests of the software system. These common modules, $M_c \subset M$, are defined as

$$M_c = \{m: M | \forall t \in T \bullet p''(t, m) > 0\}$$

All of these modules will execute regardless of the specific functionality being executed by the software system.

Yet another set of software modules may or may not execute when the system is running a particular function. These modules are said to be potentially involved modules. The set of potentially involved modules is.

$$M_p^{(f)} = \{m: M | \exists t \in T \bullet \text{EXHIBITS}(t, f) \wedge p''(t, m) > 0\}$$

In other program modules, there is extremely tight binding between the functionality and the module. That is, every time a particular function is executed, a distinct set of software modules will always be invoked. The set of indispensably involved modules is

$$M_i^{(f)} = \{m: M | \forall t \in T \bullet \text{EXHIBITS}(t, f) \Rightarrow p''(t, m) > 0\}$$

For the execution of a function, *f*, the set of modules, M_f, that are designed to implement this function is

$$M_f = M_c \cup M_p^{(f)} \cup M_i^{(i)}$$

Clearly, for the test process to be effective, the range of functionalities, *f*, must be well known.

Within a test suite, each test will exercise one or more of the system's functionalities. For a given test *t* these expressed functionalities are those with the property

$$F_e^{(t)} = \{f : F \mid \forall t \in T \bullet \text{EXHIBITS}(t, f)\}$$

For any test *t* the execution profile will consist of the probabilities $p(t, m)$ derived from

$$m \in M_t = M_c \cup M_p'(F_e^{(t)}) \cup M_i(F_e^{(t)})$$

where $M_p' = \bigcup_{F_e^{(t)}} M_p$, $M_p'(F_e^{(t)})$ and $M_i(F_e^{(t)})$ are the modules that are actually invoked in the test *t*.

From the standpoint of software design and the test process, the real problems introduced into testing are not necessarily attributable to the set of modules that are tightly bound to a functionality M_i or to the set of common modules that will be invoked for all executing processes M_c. The real problem is the set of potentially invoked modules M_p. The greater the cardinality of this set of modules, the less certain will be the outcome of a test of this functionality. For any one test of this functionality, none of the modules may execute or they may all execute.

When a program begins the execution of a functionality, we may envision this beginning as the start of a Markovian stochastic process. For the system *S* there is an activation graph that shows the transition of program control from one program module to another. To simplify this discussion, let us define a relation ACTIVATES over $M_f \times M_f$. A sample activation graph for a hypothetical program is shown in Figure 7. For example,

$$P = \begin{pmatrix} 0.7 & 0.2 & 0.1 & 0 & 0 & 0 \\ 0.4 & 0.3 & 0 & 0.3 & 0 & 0 \\ 0.1 & 0 & 0.3 & 0 & 0.1 & 0.5 \\ 0 & 0.8 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0.7 & 0 & 0 & 0.3 \end{pmatrix}$$

From this activation graph we may then construct a probability adjacency matrix, *P*, whose entries represent the transition probability from each module to another module at each epoch in the execution process. Thus, the element $p_{ij}^{(n)}$ of this matrix on the n^{th} epoch are the probabilities that $\text{ACTIVATES}(m_i, m_j)$ is true for that epoch.

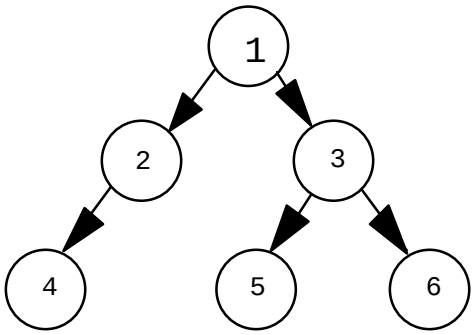


Figure 7. Hypothetical program sample activation graph.

The transition from one module to another may be seen as a stochastic process, in which case we may define an indexed collection of random variables $\{X_t\}$, where the index *t* runs through a set of non-negative integers, $t = 0,1,2,\cdots$ representing the epochs of the process. At any particular epoch the software is found to be executing exactly one of its *M* modules. The fact of the execution occurring in a particular module is a *state* of the system. For this software system, the system is found in exactly one of a finite number of mutually exclusive and exhaustive states that may be labeled $0,1,2,\cdots,M$. In this representation of the system, there is a stochastic process $\{X_t\}$, where the random variables are observed at epochs $t = 0,1,2,\cdots$ and where each random variable may take on any one of the $(M+1)$ integers, from the state space $A = \{0,1,2,\cdots,M\}$.

A stochastic process $\{X_t\}$ is a Markov chain if it has the property that

$$\Pr[X_{t+1} = j | X_t = i_t, X_{t-1} = i_{t-1}, X_{t-2} = i_{t-2}, \cdots, X_0 = i_0] = \Pr[X_{t+1} = j | X_t = i_t]$$

for any epoch $t = 0,1,2,\cdots$. and all states $i_0, i_1, \cdots, i_t$ in the state space *A*. This is equivalent to saying that the conditional probability of executing any module at any future epoch is dependent only on the current state of the system. The conditional probabilities $\Pr[X_{t+1} = j | X_t = i_t]$ are called the transition probabilities. In that this nomenclature is somewhat cumbersome, let $p_{ij}^{(n)} = \Pr[X_n = j | X_{n-1} = i]$. Within the execution of a given functionality, the behavior of the system is static. That is, the transition probabilities do not change from one epoch to another. Thus, $\Pr[X_{t+1} = j | X_t = i_t] = \Pr[X_1 = j | X_0 = i_0]$ for *i, j,* in *S*, which is an additional condition of a Markov process.

Since the $p_{ij}^{(n)}$ are conditional probabilities, it is clear that

$$p_{ij}^{(n)} \geq 0, \text{ for all } i,j \text{ in } A, \; n = 0,1,2,\cdots$$

and

$$\sum_{j=0}^{M} p_{ij}^{(n)} = 1, \text{ for all } i \text{ in } A \text{ and } n = 0,1,2,\cdots$$

If we use the nomenclature $\mathbf{P}$ to denote the matrix of one-step transition probabilities at the initial epoch, then the system at the $n+1^{st}$ epoch can be obtained from the expression

$$\mathbf{P}^{(n)} = \mathbf{P}^n = \mathbf{P} \cdot \mathbf{P}^{n-1}$$

What we would like to ascertain is the unconditional probability of being in a particular module at a particular epoch. To find this conditional probability let us first observe that

$$\Pr[X_{t+1} = j | X_t = i_t] = \Pr[X_1 = j | X_0 = i_0]$$

It is clear that the unconditional probability of executing a module *j* and epoch *n*, then, is dependent only on the initial state of the system. Thus,

$$\Pr[X_n = j] = \sum_{i=0}^{M} p_{ij}^{(n)} \Pr[X_0 = i].$$

Interestingly enough, for all software systems there is a distinguished module, the `main` program module that will always receive execution control from the operating system. If we denote this main program as module 0 then,

$$\Pr[X_0 = 0] = 1 \text{ and } \Pr[X_0 = i] = 0 \text{ for } i = 1,2,\cdots,M$$

We can see, then, that the unconditional probability of executing in a particular module *j* is

$$\Pr[X_n = j] = p_{ij}^{(n)} \Pr[X_0 = 0] = p_{ij}^{(n)}$$

The granularity of the term epoch is now of interest. An epoch is operationally defined to be the transition of a software system from one module to another. Computer programs executing in their normal mode will make state transitions between program modules rather rapidly. In terms of real clock time, many epochs may elapse in a relatively short period. Thus, we will now turn our attention to the steady-state behavior of the software system.

First, let us observe that the states being modeled represent nodes on an activation tree. As such, these states are intrinsically aperiodic. Positive recurrent states which are aperiodic are called ergodic states. If states *i* and *j* are ergodic, then it can be shown that

$$\lim_{n \to \infty} p_{ij}^{(n)} = \tau_j$$

where the τ_j's satisfy the following steady state equations:

$$\tau_j > 0,$$
$$\tau_j = \sum_{i=0}^{M} \tau_i p_{ij}^{(0)}, \text{ for } j \text{ in } A,$$
$$\sum_{j=0}^{M} \tau_j = 1.$$

The τ_j's are the steady-state probabilities of the Markov chain. These probabilities ***profile*** the behavior of the system. They may also be interpreted as stationary probabilities. That is, if $\Pr[X_0 = j] = \tau_j$ for all j in A then $\Pr[X_n = j] = \tau_j$ for any later epoch n.

Modeling Software Failures

While the Markovian process discussed earlier dealt with a program that would not fail, we now wish to examine the potential for modeling the failure of a software system. When a program module fails, we can imagine that the module has made a transition to an absorbing state, a failure state, in the Markov transition matrix. Thus, every program may be thought to have a virtual module representing the failed state of program. When this virtual module receives control, it will not relinquish it. The transition matrix for this new model is augmented by an additional row and a new column. For a program with M modules, let the error state be represented by a new state, $K = M + 1$. For this new state,

$$p_{Kj}^{(n)} \begin{cases} = 0 & \text{for all } j = 1,2,\ldots,M \\ = 1 & \text{for } j = K \end{cases}, \; n = 0,1,2,\cdots$$

This represents the augmented row of the new transition matrix. Each row in the transition matrix will be augmented by a new column entry $p_{iK}^{(n)}$ for $i = 1,2,\cdots,M$, where $p_{iK}^{(n)}$ represents the probability of the failure of the i^{th} module in the n^{th} epoch.

The Estimation of Profiles

When a program is executing a functionality it will apportion its time among a set of modules. As such it will transition from one module to the next on an activation sequence. Each module activated in this activation sequence will have an associated activation frequency. Let $p(t,m)$ represent the proportion of transitions for each of the modules that comprise the system. The particular set of M_t modules that will execute are determined by the functionalities $F_e^{(t)}$ that are expressed. If $p(t,m)$ is normalized by dividing by the total event transitions, the result is the *execution profile* for the function.

Another way to look at the execution profile is that $p(t,m) / \sum_m p(t,m)$ will represent an estimate of τ_m, the steady-state probability of the execution of module *m* in any epoch. However, as the software is subjected to a series of unique and distinct functional expressions, there will be a different Markov chain

for each of the user's operations in that each will implement a different set of functions that will, in turn, invoke possibly different sets of program modules.

Functional Profiles

When a software system is constructed by the software developer, it is designed to fulfill a set of specific functional requirements. The user will run the software to perform a set of perceived operations. In this process, the user will typically not use all of the functionalities with the same probability. The *functional profile* of the software system is the set of unconditional probabilities of each of the functionalities *F* being executed by the user. Let *Y* be a random variable defined on the indices of the set of elements of *F*. Thus, $o_k = \Pr[y = k]$ is the probability that the user is executing program functionality *k* as specified in the functional requirements of the program [Musa, 1992]. A program executing on a serial machine can only be executing one functionality at a time. The distribution of *o*, then, is multinomial for programs designed to fulfill more than two specific functions. The prior probabilities for each $f \in F$ must be known before the software is designed.

Execution Profiles

Each test suite will implement a subset of functionalities, i.e. $F_e^{(t)} \subset F$. As each test is run to completion it will generate a test *execution profile*. This test execution profile may represent the results of the execution of one or more functions. In order to simplify the discussion let us first describe a test that expresses precisely one functionality. For this test case the probability that a particular module may execute is, in fact, a conditional probability. Then $u_{ik} = \Pr[X_n = i | Y = k]$ where $k = 1, 2, \cdots, \#\{F\}$ represents the execution profile for a set of modules executing a function *k* exclusively and $\#\{F\}$ is the cardinality of the set of functions. As was the case with the functional profile, the distribution of the execution profile is also multinomial for a software system consisting of more than two modules. As a matter of the design of a program, there may be a non-empty set $M_p^{(f)}$ of modules that may or may not be executed when a particular functionality is exercised. This will, of course, cause the cardinality of the set M_f to vary. A particular execution may not invoke any of the modules of $M_p^{(f)}$. On the other hand, all of the modules may participate in the execution of that functionality. As we will see, this variation in the cardinality of M_f within the execution of a single functionality will contribute significantly to the amount of test effort that will be necessary to test such a functionality.

Most tests, though, do not exercise precisely one functionality. Rather, they may apportion time across a number of functionalities. For a given test, let *l* be a proportionality constant equal to the proportion of time spent executing a given function. Thus, $0 \le l_k \le 1$ will represent the proportion of time executing the k^{th} functionality in $F_e^{(t)}$. Thus the test execution profile will represent a linear combination of the conditional probabilities, u_{ik} as follows:

$$p_i = \sum_{f_k \in F_e^{(t)}} l_k u_{ik}.$$

Module Profiles

The manner in which a program will exercise its many modules as the user chooses to execute the functionalities of the program is determined directly by the design of the program. Indeed, this mapping of functionality onto program modules is the overall objective of the design process. The *module profile*, *q*, is the unconditional probability that a particular module will be executed based on the design of the program. It is derived through the application of Bayes' rule. First, the joint probability that a given module is executing and the program is exercising a particular function is given by

$$\Pr[X_n = j \cap Y = k] = \Pr[Y = k]\Pr[X_n = j | Y = k] = o_k u_{ik}$$

Thus, the unconditional probability, q_i of executing module *i* under a particular design is

$$\begin{aligned} q_i &= \Pr[X_n = i] \\ &= \sum\nolimits_k \Pr[X_n = i \cap Y = k] \\ &= \sum\nolimits_k o_k u_{ik} \end{aligned}$$

As was the case for the functional profile and the execution profile, only one module can be executing at any one time. Hence, the distribution of *q* is also multinomial for more than two modules.

Dynamic Profile Execution

One of the objectives in modeling the reliability of a system is to come to understand the nature of the distribution of the probabilities for various profiles. We have so far come to recognize these distributions in terms of their multinomial nature. This distribution is useful for representing the outcome of an experiment involving a set of mutually exclusive events. Let $S = \bigcup_{i=1}^{M} S_i$ where S_i is one of *M mutually* exclusive sets of events. Each of these events would correspond to a program executing a particular module in the total set of program modules. Further, let $\Pr(S_i) = w$ and

$$w_T = 1 - w_1 - w_2 - \cdots - w_M,$$

under the condition that $T = M + 1$, as defined earlier. In which case w_i is the probability that the outcome of a random experiment is an element of the set S_i. If this experiment is conducted over a period of *n* trials then the random variable S_i will represent the frequency of S_i outcomes. In this case, the value, *n*, represents the number of transitions from one program module to the next. Note that

$$X_T = n - X_1 - X_2 - \cdots - X_M$$

This particular distribution will be useful in the modeling of a program with a set of *k* modules. During a set of *n* program steps, each of the modules may be executed. These, of course, are mutually exclusive events. If module *i* is executing then module *j* cannot be executing.

The multinomial distribution function with parameters n and $\boldsymbol{w} = (w_1,w_2,\ldots,w_T)$ is given by

$$f(\boldsymbol{x}|\, n,\boldsymbol{w}) = \begin{cases} \dfrac{n!}{x_1!x_2!\cdots x_M!} w_1^{x_1} w_2^{x_2} \ldots w_M^{x_M}, & (x_1, x_2, \cdots, x_M) \in S \\ 0 \quad \text{elsewhere} \end{cases}$$

where x_i represents the frequency of execution of the i^{th} program module.

The expected values for the x_i are given by

$$E(x_i) = \bar{x}_i = nw_i\ , \quad i = 1,2,\ldots,k\ ,$$

the variances by

$$Var(x_i) = nw_i(1 - w_i)$$

and the covariance by

$$Cov(x_i, x_j) = -nw_i w_j\ , \quad i \neq j$$

We would like to come to understand, for example, the multinomial distribution of a program's execution profile while it is executing a particular functionality. The problem, here, is that every time a program is run we will observe that there is some variation in the profile from one execution sample to the next. It will be difficult to estimate the parameters $\boldsymbol{w} = (w_1,w_2,\ldots,w_T)$ for the multinomial distribution of the execution profile. Rather than estimating these parameters statically, it would be far more useful to us to get estimates of these parameters dynamically as the program is actually in operation.

To aid in the process of characterizing the nature of the true underlying multinomial distribution, let us observe that the family of Dirichlet distributions is a conjugate family for observations that have a multinomial distribution [Wilks, 1962]. The p.d.f. for a Dirichlet distribution, $D(\boldsymbol{\alpha};\alpha_T)$, with a parametric vector $\boldsymbol{\alpha} = (\alpha_1, \alpha_2,\ldots, \alpha_M)$ where $(\alpha_i > 0;\ i = 1,2,\ldots, M)$ is

$$f(\boldsymbol{w}|\boldsymbol{\alpha}) = \frac{\Gamma(\alpha_1 + \alpha_2 + \cdots + \alpha_M)}{\Gamma(\alpha_1)\Gamma(\alpha_2)\cdots\Gamma(\alpha_M)} w_1^{\alpha_1 - 1} w_2^{\alpha_2 - 1} \cdots w_M^{\alpha_M - 1}$$

where $(w_i > 0;\ i = 1,2,\ldots, M)$ and $\sum_{i-1}^{T} w_i = 1$. The expected values of the w_i are given by

$$E(w_i) = \mu_i = \frac{\alpha_i}{\alpha_0}$$

where $\alpha_0 = \sum_{i=1}^{T} \alpha_i$. The variance of the x_i is given by

$$Var(w_i) = \frac{\alpha_i(\alpha_0 - \alpha_i)}{\alpha_0^2(\alpha_0 + 1)},$$

and the covariance by

$$Cov(w_i, w_j) = -\frac{\alpha_i \alpha_j}{\alpha_0^2(\alpha_0 + 1)}$$

Within the set of expected values $\mu_i, i = 1, \cdots, T$, not all of the values are of equal interest. We are interested, in particular, in the value of μ_T. This will represent the probability of a transition to the terminal failure state from a particular program module. So that we might use this value for our succeeding reliability prediction activities, it will be useful to know how good this estimate is. To this end, we would like to set 100α% confidence limits on the estimate. For the Dirichlet distribution, this is not clean. To simplify the process of setting these confidence limits, let us observe that if $\boldsymbol{w} = (w_1, w_2, \ldots, w_M)$ is a random vector having the M-variate Dirichlet distribution, $D(\boldsymbol{\alpha}; \alpha_T)$, then the sum $z = w_1 + \cdots + w_M$ has the beta distribution,

$$f_\beta(z|\gamma, \alpha_T) = \frac{\Gamma(\gamma + \alpha_T)}{\Gamma(\gamma)\Gamma(\alpha_T)} z^\gamma (1-z)^{\alpha_T}$$

or alternately

$$f_\beta(w_T|\gamma, \alpha_T) = \frac{\Gamma(\gamma + \alpha_T)}{\Gamma(\gamma)\Gamma(\alpha_T)} (1-w_T)^\gamma (w_T)^{\alpha_T},$$

where $\gamma = \alpha_1 + \alpha_2 + \cdots + \alpha_M$.

Thus, we may obtain 100α% confidence limits for

$$\mu_T - a \le \mu_T \le \mu_T + b$$

from

$$F_\beta(\mu_T - a|\gamma, \alpha_T) = \int_0^{\mu_T - a} f_\beta(w_T|\gamma, \alpha_T) dw_T = \alpha/2$$

and

$$F_\beta(\mu_T + b|\gamma, \alpha_T) = \int_0^{\mu_T + b} f_\beta(w_T|\gamma, \alpha_T) dw_T = 1 - \alpha/2$$

Where this computation is inconvenient, let us observe that the cumulative beta function, F_β, can also be obtained from existing tables of the cumulative binomial distribution, F_b, by making use of the knowledge from Raiffa [1961] that

$$F_b(\gamma|\mu_T - a, \gamma + \alpha_T) = F_\beta(\mu_T - a|\gamma, \alpha_T)$$

and

$$F_b(\alpha_T \mid 1-(\mu_T + b), \gamma + \alpha_T) = F_\beta(\mu_T + b \mid \gamma, \alpha_T),$$

The value of the Dirichlet conjugate family for modeling purposes is twofold. First, it permits us to estimate the probabilities of the module transitions directly from the observed transitions. Secondly, we are able to obtain revised estimates for these probabilities as the observation process progresses. Let us now suppose that we wish to model the behavior of a software system whose execution profile has a multinomial distribution with parameters *n* and $\boldsymbol{W} = (w_1, w_2, \cdots, w_M)$ where *n* is the total number of observed module transitions and the values of the w_i are unknown. Let us assume that the prior distribution of $\boldsymbol{W}$ is a Dirichlet distribution with a parametric vector $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_M)$ where $(\alpha_i > 0; i = 1,2,\cdots, M)$. Then the posterior distribution of $\boldsymbol{W}$ for the behavioral observation $\boldsymbol{X} = (x_1, x_2, \cdots, x_M)$ is a Dirichlet distribution with parametric vector $\alpha^* = (\alpha_1 + x_1, \alpha_2 + x_2, \cdots, \alpha_M + x_M)$ [DeGroot, 1970]. As an example, suppose that we now wish to model the behavior of a large software system with such a parametric vector. As the system makes sequential transitions from one module to another, the posterior distribution of $\boldsymbol{W}$ at each transition will be a Dirichlet distribution. Further, for *i* = 1,2,…,*T* the i^{th} component of the augmented parametric vector $\boldsymbol{\alpha}$ will be increased by 1 unit each time module m_i is executed.

Functional Reliability

Not all states that a system of *K* program modules can get into hold equal fascination for us. In the augmented program model above, a virtual program module, m_K, was employed to represent a program module representing an error state. From the steady-state equations of the Markov model of this system,

$$\begin{aligned} &\tau_j > 0, \\ &\tau_j = \sum_{i=0}^{K} \tau_i p_{ij}, \ \text{for} j \ \text{in} A \ , \\ &\sum_{j=0}^{K} \tau_j = 1, \end{aligned}$$

the probability $\tau_K^{(f)}$ is of interest in that this is the long-term probability of winding up in state *K*, the virtual failure module when executing the function, *f*. Quite simply, the reliability of the system executing a given functionality, *f*, is given by $r^{(f)} = 1 - \tau_K$. Further, with the proper instrumentation in a program, the reliability of this system may be estimated *on the fly* by exploiting the Dirichlet conjugate distribution family for the multinomial distribution of execution probabilities.

This reliability estimate, however, is only for a particular functionality. It is based entirely on the execution profile of a given functionality. Thus, each function has its own independent reliability

assessment. That was the original intent of this investigation, to show that reliability is dependent directly on the function that is executing. It is program functions that fail. Some functions are more reliable than others.

System Reliability

From a more global perspective, it is also useful to consider the reliability of the software as a system of functions. To this end, let us observe that not all functionalities will operate with equal probability. There is a functional profile, $o_k = \Pr[y = k]$, which is the probability that the user is executing program functionality *k* as specified in the functional requirements of the program. The reliability of the system, r^S, is the expected value of the functional reliability, to wit:

$$r^S = E(\mathbf{r}) = \sum_{i=1}^{\#\{F\}} o_i r_K^{f_i}$$

where $\#\{F\}$ is the cardinality of the set of functions and $r_K^{f_i}$ is the estimate of the reliability of the i^{th} function. The variation in reliability from one function to another will have an impact on the understanding of the system reliability. The overall system reliability will be bounded below, conceptually, by the least reliable of the program functionalities.

User Guidelines for Data Collection

To obtain the necessary information for the posterior distributions of the execution profiles, we will need the program (or operating system [or hardware]) to maintain certain information for us. Specifically, for the computation of the posterior distributions of the Dirichlet distributions for each of the functionalities, we will need a vector containing a simple count of the number of times that each module has been executed. Thus the complete information necessary for this modeling process will be kept in an $n \times m$ transition matrix, $\boldsymbol{T}$ for a program with *n* functions and *m* modules. The elements, t_{ij}, of this matrix represent the number of times module *j* has been activated while executing function *i*.

Necessarily, if the program is maintaining the transition matrix, it will not be able to update the vector for the virtual module that represents the failure state of the program. When the program transitions to this hypothetical state, it will be dead. If $\boldsymbol{T}$ is being maintained by the operating system, or even better, in the hardware, then we will also be able to capture the program's transition into the virtual failure module.

In the specific case of the Space Shuttle PASS, there now exists a processor (PMIP) that will serve the data collection process very well. In this sense the Shuttle code is already instrumented for reliability assessment. To develop the system reliability for PASS, the functional reliabilities must first be established. To achieve this objective, tests must be identified that express each of the functionalities we identified in Milestone I Phase II. For each of the tests, an execution profile must be constructed. This will be used to derive the state transition probabilities for the transition matrix of the basic Markov model.

The reliability augmented model may be constructed by adding the estimated state transition probabilities from each of the functional states to the virtual failure state. The information to reconstruct these transition probabilities may, in fact, not be available. In this case, the prior probabilities may be set at a realistically small value representing the intuitive reliability of each of the functions. The more accurate these prior probabilities are, the more rapidly the Markov process will converge to reasonably accurate and useful reliability figures. As is always the case with the Bayesian approach, if we do not have any information as to the prior probabilities, we may assume that all events are equiprobable. Thus, if there are ten modules in the activation set for a particular function, then each would have a prior probability of 0.1 under this assumption of equiprobable outcomes.

Specific Data Collection Items

1. A state vector that will track the specific function that a program is currently executing. This vector will contain sufficient information to identify the function being executed. For example, this vector might contain zeros for all entries except for the i^{th} entry which would be a one. This would indicate that function *i* is currently executing.

2. As a specific function is executing, it will have an associated activation tree of *m* program modules. The program will transfer control from one program module to another. As control passes from module *i* to module *j*, this fact will be used to update a transition matrix. The transition matrix $\boldsymbol{M}$ will initially have zeros in all entries. As control passes for module *i* to *j* the matrix entry m_{ij} will be augmented by one. As the system switches from one functionality to another, it will also do a context switch on the function transition matrix. The transition frequencies recorded in this matrix will be used to compute the execution profiles for each functionality.

3. A failure event may be represented by two different states in this matrix. First, there will be a module that will serve as a total failure of the system. When control is apparently transferred to this module, it means that the program has gone into an irretrievable collapse. The second failure state will be represented by a module category representing a contingency management routine. Such a contingency as a floating point overflow might be raised that will not result in the complete failure of the program. This is a recoverable failure. Both failure states will appear as module entries in the transition matrix $\boldsymbol{M}$. Thus, if the program has a total of *m* modules then the augmented matrix will have $m + 2$ states.

4. The complete failure management process data will be stored in an $n \times (m + 2)$ transition matrix containing the current steady-state probabilities of the Markov process.

5. The above data may be periodically logged by the operation system or programming environment to secondary memory. The contents of these registers and matrices constitute the complete operating environment of the program as it executes. The precise state of software system failures may be captured from the contents of these data matrices and registers.

6. The current reliability of each of the system functions may be determined from the last two columns of the transition matrix. This may be displayed in real time on a console using either a digital or an analog performance meter. The scale of this meter may be uniform on the interval from least reliable to most reliable or it may be logarithmic for ultra-reliable software

Reliability Modeling With Modules of Differing Ages

All of the OPS that comprise the Space Shuttle PASS contain software modules of wildly differing ages. Many of the program modules are over 10 years old. Some modules are new to OI24. Some of the legacy code has been modified to varying degrees over the last several OIs. For standard approaches to reliability modeling, this system of modules of varying ages presents a real challenge. Under this new approach, time is not a term in the model. It really doesn't matter how young or old a HAL code module is. The principal difference, under this modeling approach, between an element of legacy code and a junior program module is the determination of the prior probabilities for the Direchlet distribution. For legacy code, the priors for the distribution will be the posterior probabilities from the previous software build (OI). The choice of the priors for modules that are new to the system or have been modified may best be derived from the functional complexities of these modules under software testing. These prior failure probabilities will be developed from historical failure data.

Summary

The failure of a software system is dependent only on what the software is currently doing. If a program is currently executing a functionality that is expressed in terms of a set of fault-free modules, this functionality will certainly execute indefinitely without any likelihood of failure. If a program executes a sequence of fault-free modules, it will not fail here either. A failure event can only occur when the software system executes a module that contains faults. Even in this event, the faults must lie in a region of the code that is likely to be expressed during the execution of a function. If a functionality is never selected that drives the program into an activation subtree that contains faults, then the program will never fail. Alternatively, a program may well execute successfully in an activation subtree that contains faults just as long as the faults are not expressed.

Some of the problems that have arisen in past attempts at software reliability determination all relate to the fact that the perspective has been distorted. Programs do not wear out over time; if they are not modified, they will certainly not improve over time, nor will they get less reliable over time. The only thing that really impacts the reliability of a software system is what the system is doing at the moment. A program may work very well for a number of years based on the functions that it is asked to execute. This same program may suddenly become quite unreliable if its functionality is changed by the user. By keeping track of the state transitions from module to module and function to function we may learn exactly where a system is fragile. This information coupled with the functional profile will tell us just how reliable the system will be when we use it as specified.

With this new approach to reliability modeling, the reliability estimation process need no longer be a laborious and external function to the operation of the PASS software. An objective of this study is to outline a procedure whereby the PASS software may be instrumented to provide a dynamic assessment of the reliability of the individual functions that constitute the system and also the reliability of the PASS as a whole. With the appropriate software instrumentation of future releases of the Space Shuttle PASS, there is every reason to believe that the system will be able to provide an accurate and dynamic assessment of its own reliability that may be displayed in real time on a system console.

Section III
HAL/S Metric Analyzer, Rev. 3.1

Introduction

The objective of this deliverable is to document the latest release of HALMet and its companion software tool, PCA-RCM. The installation and operating manuals for release 3.1 of HALMet are included as Appendixes A and B. The installation and operating instructions for the PCA-RCM tool are included as Appendixes C and D.

This report documents the modifications in the latest revision of the software metrics system for the Space Shuttle PASS. This final version has been enhanced to execute on the SUN workstation in the NASA environment. It also includes a software tool that will perform principal components analysis on the raw complexity measures to produce domain metrics. An associated tool for the computation of system-relative complexity has been incorporated into this tool.

HALMet 3.1

The objective of the HALMet software measurement tool is to obtain measurements on 24 software complexity measures for programs written in HAL/S. The 24 metrics that have been selected for use in this tool are shown in Table 7. Of the 24 metrics chosen, 20 were found to describe a significant amount of variation in the total faults in the Space Shuttle PASS historical database for discrepancy reports (DRs). Four of the metrics designed to measure the real-time quality of the software did not contribute significantly to the understanding of software faults in the currently available HAL/S source code. There is reason to believe that they might play a stronger role in the measurement of earlier systems. For this reason, the following four real-time metrics are calculated by the HALMet software measurement tool, but are removed from further analysis: *Signals, Terms, SigA, and TermA.* For information on installation and operation of the tool, see Appendixes A and B.

Table 7. Metrics and Their Definitions

η_1	Halstead's count of the number of unique operators
η_2	Halstead's count of the number of unique operands
N_1	Halstead's count of the total number of operators
N_2	Halstead's count of the total number of operands
Stmt	Count of total non-commented source statements
LOC	Count of the total number of non-commented lines of code
Comm	Count of the total number of commented source statements
Nodes	Number of nodes in the control flow graph
Edges	Number of edges in the control flow graph
Paths	Number of distinct paths through the control flow graph
Cycles	Count of the number of cycles in the control flow graph
MaxP	Total arc length of the maximum path through a control flow graph
AveP	Average path length of the total paths in a control flow graph
DataStruc	Measure of data structure complexity
Sets	Number of `SET` statements in a program module
Resets	Number of `RESET` statements in a program module
Signals	Number of `SIGNAL` statements in a program module
Cans	Number of `CANCEL` statements in a program module
Terms	Number of `TERMINATE` statements in a program module
SetA	Largest number of arguments in any one `SET` statement
ResA	Largest number of arguments in any one `RESET` statement
SigA	Largest number of arguments in any one `SIGNAL` statement
CanA	Largest number of arguments in any one `CANCEL` statement
TermA	Largest number of arguments in any one `TERMINATE` statement

System Baselining

The measurement of an evolving software system through the shifting sands of time is not an easy task. Perhaps one of the most difficult issues relates to the establishment of a baseline against which the evolving systems may be compared. This problem is very similar to that encountered by the surveying profession. If we were to buy a piece of property, there are certain physical attributes that we would like to know about that property. First, how big is the property? What is its physical shape? Where is it? What is its

physical elevation? What about the physical topology of the property? In this example, the first two questions may be answered with a transit and a measuring tape at the site. To answer the questions as to the location or the elevation of the property, we must have something more, since we cannot make these determinations from the site alone. We will have to seek out a benchmark. The benchmark represents a survey marker that represents a point in a larger standard grid wherein each point is clearly related to every other point in the grid both in terms of distance and elevation. This benchmark may be some distance from our property. To continue this analogy, to measure the topology of the property, we must first establish a fixed point or baseline on the property. The distance and the elevation of every other point on the property may then be established in relation to the fixed baseline. Interestingly enough, we can pick any other point on the property, establish a new baseline, and get exactly the same topology for the property. The property does not change. Only our perspective changes.

Baselining a System

With all of this in mind, we find that the software measurement process is very much the same as the survey process. We wish to understand the individual elements of the whole system in relation to each other. We also wish to understand just how a system has evolved over time. It is very difficult to use raw complexity metrics for either of these purposes. The dilemma confronted by those who wish to use measurement of evolving software systems may be seen in Table 8. In this table we see two program modules named A and B. We have two measurements on each of these two modules: lines of code, *LOC*, and cyclomatic complexity, *V(g)*. Measurements have been taken at times T1 and T2. First, let us look at the two modules A and B at time T1. Is module A more complex than B? Now, let us look at how the system containing modules A and B has changed from time T1 to time T2. Is the system more complex at time T2 than T1? Clearly, the total number of lines of code has dropped by 10 from T1 to T2. But, the total cyclomatic complexity has risen from 35 to 37.

Table 8. A Measurement Example

	Time 1		Time 2	
Metric	Module A	Module B	Module A	Module B
LOC	200	250	210	230
V(g)	20	15	19	18

The whole notion of establishing a baseline system will allow us to begin to answer the questions raised in the dilemma created by the data represented in Table 8. The first thing that we must do is to identify

common sources of variation among the metrics. We will use principal components analysis to create a set of orthogonal measures for the software modules, all of which will be defined on the same scale. From these common domain metrics, we then have the option of computing relative complexity values for each of the program modules. This will reduce the dimensionality of the complexity problem to one single measure for each program.

When a number of successive system builds (OIs) are to be measured, we will choose one of the systems as a baseline system. All others will be measured in relation to the chosen system. This is exactly analogous to the selection of an arbitrary point or a piece of property to begin a topological survey. Sometimes it will be useful to select the initial system build for this baseline. If we select this system, then the measurements on all other systems will be taken in relation to the initial system configuration.

Data Transformation

Having established the need for a baselining approach to the analysis of a number of related systems, we realize that for measurement purposes, it will be necessary to standardize all of the raw metrics so that they are on the same relative scale. For the i^{th} module, m_i^j, on the j^{th} build of the system, there will be a data vector, $\boldsymbol{x}_i^j = < x_{i1}^j, x_{i2}^j, \ldots, x_{ik}^j >$, of *k* raw complexity metrics. We can standardize each of the raw metrics by subtracting its mean, $\bar{x}_1^j$, and dividing by its standard deviation, δ_1^j, such that $z_{1i}^j = \frac{x_{1i}^j - \bar{x}_1^j}{\delta_1^j}$ represents the standardized value of the first raw metric for the i^{th} module on the j^{th} build. The problem with this method of standardizing is that it will erase the effect of trends in the data. For example, let us assume that we were taking measurements on *LOC* and that the system we were measuring grew in this measure over successive builds. If we were to standardize each system by its own mean *LOC* and its own standard deviation, then the mean of this system would be zero. Thus, we will standardize the raw metrics in relation to the baseline system such that the standardized metric vector for the i^{th} module, m_i^j, on the j^{th} build would be $\boldsymbol{w}_i^j = \frac{\boldsymbol{x}_i^j - \bar{\boldsymbol{x}}_i^0}{\boldsymbol{\sigma}_i^0}$ where $\bar{\boldsymbol{x}}_i^0$ is a vector containing the means of the raw metrics for the baseline system, and $\boldsymbol{\sigma}_i^0$ is a vector of standard deviations of these raw metrics. Thus, for each system, we may build an $m \times k$ data matrix, $\boldsymbol{W}^j$, that contains the standardized metric values relative to the baseline system.

Using the standardized data matrix, $\boldsymbol{W}^j$, the orthogonal domain metrics for each of the systems will also be developed in relation to the baseline system. Let $\boldsymbol{T}$ represent the $k \times o$ transformation matrix for the computation of the domain metrics for the baseline system where *o* represents the number of orthogonal complexity domains in the baseline data. The domain metrics for each of the remaining systems may then be derived from $\boldsymbol{T}$ as follows: $\boldsymbol{D}^j = \boldsymbol{W}^j \boldsymbol{T}$, where $\boldsymbol{D}^j$ is an $m \times o$ matrix containing the baselined domain metrics for the *m* program modules on the j^{th} build. An example of the transformation matrix, $\boldsymbol{T}$, can be seen in Table 9.

Table 9. Transformation Matrix for PASS Software

Variable	Domain1	Domain2	Domain3	Domain4
η_1	0.069	0.031	-0.076	-0.056
η_2	0.086	-0.005	-0.168	-0.058
N_1	0.087	0.005	-0.218	-0.042
N_2	0.087	0.009	-0.221	-0.034
Stmt	0.090	-0.010	-0.199	-0.034
LOC	0.089	0.037	-0.125	0.004
Comm	0.078	-0.013	-0.173	-0.048
Nodes	0.085	-0.100	0.140	0.042
Edges	0.084	-0.111	0.145	0.043
Paths	0.066	-0.122	0.089	0.014
Cycles	0.059	-0.154	0.259	0.104
MaxP	0.083	-0.123	0.224	0.083
AveP	0.082	-0.121	0.234	0.089
DataStruc	0.071	-0.063	-0.119	-0.038
Sets	0.053	0.240	0.122	-0.134
Resets	0.043	0.246	0.185	-0.163
Cans	0.032	0.221	-0.011	0.511
SetA	0.048	0.209	0.133	-0.223
ResA	0.037	0.231	0.158	-0.266
CanA	0.028	0.219	-0.012	0.517
Eigenvalues	9.963	2.442	1.844	1.357

In the computation of the principal components for the baseline data, each of the resulting orthogonal complexity domains has an associated eigenvalue, λ_i. We can now use these baselined eigenvalues to compute the relative complexity of each of the system builds as follows: $\boldsymbol{\rho}^j = \boldsymbol{\Lambda D}^j$ where $\boldsymbol{\rho}^j$ is a vector of relative complexity values for the *m* modules of the j^{th} build, and $\boldsymbol{\Lambda}$ is the vector of eigenvalues associated with the *o* domains of $\boldsymbol{D}^j$.

Insertions and Deletions

In light of all of these carefully planned transformations, there is still one significant problem that occurs in evolving software systems with which we have not yet dealt. That is the fact that software modules come and go. Stated another way, the cardinality of the module set will certainly vary over time. This is not a problem when it comes to the computation of the individual module domain metrics and the computation of module relative complexity. It is a problem when we are looking at the average system metrics. For example, if the initial build of a system contained *m* program modules and the next system contains *m* + 1 modules, how should the average relative complexity of the new system be established? We can understand this problem a little better if we consider a program module that was simply split into two modules from the first to the second build. This being the case, the relative complexity of each of the two new modules will be less than the relative complexity of the parent module. Thus, if we were to compute the average relative complexity of the new system with the value of *m* + 1 as the normalizing value, then the apparent complexity of the new system will have been reduced. However, because of the coupling complexity introduced between the two new modules, the net system complexity will have risen. To this end, the normalizing value for the computation of all averages will be the cardinality of the set of modules in the baseline system.

By definition, the average relative complexity, $\boldsymbol{\rho}$, of the baseline *system* will be

$$\boldsymbol{\rho}^1 = \frac{1}{N^1}\sum\nolimits_i \rho_i^{v_i^1} = 50,$$

where N^1 is the cardinality of the set of program modules. As the system progresses through a series of builds, system complexity will tend to rise. Thus, the system relative complexity of the k^{th} version of a system may be represented by a non-decreasing function of module relative complexity as follows:

$$\boldsymbol{\rho}^k = \frac{1}{N^1}\sum\nolimits_i \rho_i^{v_i^k},$$

where v_i^k represents an element from the configuration vector, $\boldsymbol{\nu}^k$, described earlier. This change in the overall relative complexity of the system over time is represented pictorially in Figure 8. This is an example of the relative complexity of the most recent 20 software builds for the Space Shuttle PASS. For this presentation, the baseline system is represented by system 0 on the X axis of this graph and, in this case, corresponds to OI23.01. All other systems are measured relative to this one. However, it is important to keep in mind that while there may have been some very good reasons for choosing this particular OI as the baseline, the user is free to choose any OI that is deemed appropriate for this purpose as pertaining to the above discussion. One pattern that becomes obvious from this figure is that the complexity of a system will continue to rise over the life of the software system. If we were to move the baseline system back another 10 systems in time, the general upward trend of the complexity of the system

would be sustained. The particular baseline for this figure was selected because of the change activity that we had observed before and after this baseline build.

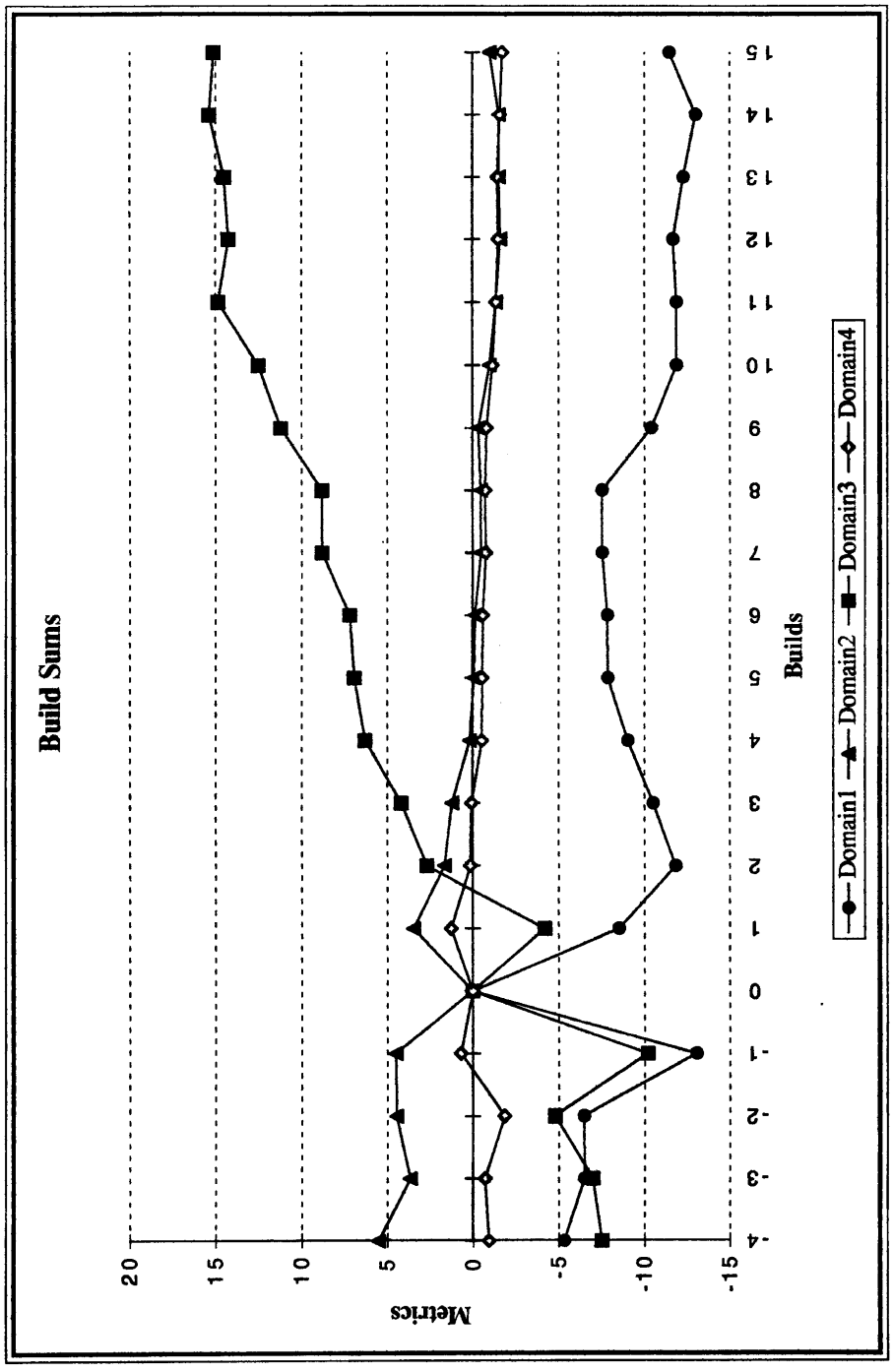


Figure 8. Change in overall relative complexity of system.

Principal Components Analysis/Relative Complexity Metric (PCA-RCM) Tool 2.0

The objective of the PCA-RCM data analysis tool is, first, to perform a PCA on the specified input file containing raw metrics data and, second, to use the output of this analysis to calculate the RCM for the observations in the original raw metrics data file. The process by which this is achieved, together with the resultant RCM calculations, is dependent on the type of analysis desired. The requested analysis is assumed to be either a baseline analysis or a build analysis as discussed in the previous sections. The type of analysis to be performed along with the desired input and output files can be specified on the command line.

One important fact to keep in mind concerning the applicability of this tool is that this tool in no way requires the existence or prior use of the previously mentioned HALMet tool. Although the PCA-RCM tool was designed to be used in conjunction with the HALMet tool, this dependence only exists in the sense that the HALMet tool produces an output file that is formatted in such a way as to be directly suitable as input to the PCA-RCM tool. In other words, the ability to use the PCA-RCM tool is completely independent of the HALMet tool. The only requirement for proper operation of the PCA-RCM tool is that the input raw metrics data file be properly formatted. The proper format consists of each line of the input file being composed of a name field followed by any number of numeric fields. The number of fields of each line is assumed to be the same as the number of fields occurring in the first line with the name field being the first field of each line. The field separator is assumed to be 'white space' (spaces or tabs), while a line feed is the character separating the lines.

Proper use of the PCA-RCM tool requires that the user not only be using a correctly formatted input file, but also that he be aware of the operation of the tool as it pertains to the specific type of analysis chosen via the '-base' or '-build' command-line option. Typically, the first analysis to be performed will be a baseline analysis, which is specified as being the desired analysis by using the '-base' command-line option. Under this type of analysis, a full PCA is performed, and certain relevant information that will be necessary for subsequent analyses is stored for later use. This relevant information consists of the column means and standard deviations, the eigenvalues, and the transformation matrix.

Having performed the initial baseline analysis, a number of subsequent analyses may be performed relative to this initial baseline. At this point, the user would have already determined the relevancy of this baseline as it pertains to the new data file that is being analyzed by, for example, its place in the sequential order of the data files to be analyzed. By virtue of this premise, the previously stored baseline information is loaded and used for the subsequent analysis as specified to be a build analysis using the '-build' command-line option. This analysis, therefore, is not comprised of a full PCA, but rather by a transforming of the input data file using the data generated by the previously executed baseline analysis.

This transformation is achieved in three steps. First, the given input data file is standardized using the stored means and standard deviations from the baseline analysis. Secondly, the resultant standardized data file is transformed into the domain metrics using the baseline's transformation matrix. And, finally, the RCM is calculated using the eigenvalues calculated by the PCA of the baseline data file. Through the use of this three-step process, the build analysis produces results that are relative to the baseline and, therefore, can be compared to the baseline.

For more information on the installation and operation of this tool, see Appendixes C and D.

Tool Installation and Operation Manuals

The tool package manuals for HALMet 3.1 and PCA-RCM 2.0 can be found in Appendixes A through D. The installation manuals provide a detailed explanation of both the hardware and software requirements, and they take the user through a step-by-step process for performing a full installation of all of the tools available in each tools package. The operation manuals provide a detailed description of each of the tools available in each package in terms of the following categories: tool name, description, synopsis, input, output, compilation, files used, and see also. The operation manuals are purposely designed to coincide with the manual page documents provided under the full Unix distribution. In this way, specific information pertaining to the usage of the tools can be obtained on-line.

Section IV
Functional Complexity and Test Efficiency

Introduction

A significant problem with modern software testing procedures is that the objectives of the test process are not clearly specified and sometimes not clearly understood. If questioned, most software testers will associate the testing process with a process of finding faults in programs. If this is the case, is the purpose of testing to find all of the faults? Some of the faults? How will we know when to stop testing? An implicit objective of the deterministic school of testing is to design some sort of a systematic and deterministic test procedure that will guarantee sufficient test exposure for the random faults distributed throughout a program. However, there is reason to believe that it is possible to identify measurable program attributes such as complexity that can explain a large proportion of the variance observed in the location of software faults in software modules. This being the case, it would only seem reasonable to look for software faults in the complex code segments.

We have developed techniques that will serve as a filter for the code to identify regions of the code that are most likely to contain faults. We must also come to accept the fact that some faults will always be present in the code. The objective of the testing process is simply to find those faults that will have the greatest impact on the safety/survivability of the code. Under this new view of the software testing process, the act of testing may be thought of as conducting an experiment on the behavior of the code under typical execution conditions. We will determine, a priori, exactly what we wish to learn about the code in the test process and conduct the experiment until this stopping condition has been reached.

Faults are introduced in code by systematic errors committed by programmers. With this fact in mind we have identified a set of measurable software attributes that are distinctly related to the conditions that lead to faults. Indeed, these complexity metrics have been shown to be distinctly associated with software faults. The main point is that, if we can identify those software attributes that are associated with faults, we may use these data to identify regions of code of software currently under development and test that are likely to contain faults .

One of the fundamental tenets of the statistical approach to software test is that it is possible to create fault surrogates. While we cannot know the numbers and locations of faults, we can, over time, build models based on observed relationships between faults and some other measurable software attributes. Software faults and other measures of software quality can be known only at the point the software has finally been retired from service. Only then can it be said that all of the relevant faults have been isolated and removed from the software system. On the other hand, software complexity can be measured very early in the software life cycle. Some of these measures are very good leading indicators of potential software faults. The first step in the software testing process is to construct suitable surrogate measures for software faults.

In the case of this study, a single measure, relative complexity, will be constructed to serve as such a surrogate. In those program modules that have large values of this surrogate measure, there is reasonable evidence to support the conclusion that the number of faults will also be large. If the code is made to execute a significant proportion of time in modules of large relative complexity, then the potential exposure to software faults will be great and, thus, the software will be exposed to these faults.

There are two distinct and separable issues now. First, it is one thing to have a software system that contains faults. Second, it is quite another for the regions of code containing faults to be executed. If there does not exist a set of input data that will drive a program into a region of code that is likely to contain faults, then there will probably be few failures during the processing of this input set. If, on the other hand, most every input data set will drive the code into the faulty code regions, then the software will prove quite failure prone. In this current investigation we will examine the problem of executing the code segments that contain faults for testing purposes. We will measure the execution of the suite of code modules that constitute the whole program in terms of the total proportion of time that we must spend in each program module. In this guise we will measure our exposure to potential program failures in terms of the degree of exposure to potentially faulty program modules. Thus, we may come to establish a dynamic measure of program complexity. In both of these cases, we will understand that our exposure to program faults is directly proportional to the complexity of the code that we will execute. It is one thing for a program module to be complex and thus be fault-prone; it is quite another for this module to be executed during a nominal flight scenario. If a module is complex and therefore fault-prone but not likely to execute on a nominal scenario then it will not be likely to contribute to the failure of the software system. The extent to which a software module is actually executed in a particular scenario is described in the execution profile of the system (c.f. Section II, Section VI).

The following empirical analysis is intended to be a proof of concept test. The two software systems that will be compared in this study are OI23.10 and OI24.09. Technically, the results should have been obtained from regression tests on precisely the same functions (tests) on two sequential builds (OIs). However, because it was not possible to obtain data from the testing process with the PMIP processor running, the test process will be emulated by the analysis of the two systems for which PMIP data were available.

This section represents the results of the fourth project milestone for this fiscal year. Execution profile data for PASS running on a number of different OPS provided the basis for the computation of functional complexity. An assessment of the variability of the functional complexity of the system is presented. This will show the range of functional behavior for PASS in which it is executing a spectrum of functionalities. This is important in that the ultimate reliability modeling capability for PASS is dependent on the measurement of the range of functional complexity that the system may generate as it executes a representative suite of its possible functionalities. Profile data were on a test suite for OI24.09 that corresponds directly to the same test suite for OI23.10. These data were used to emulate a regression test

of the OI24.09 to demonstrate the concept of regression test efficiency. The test efficiency of these regression tests will then be reported. This will serve as a demonstration of the concept in which the measurement process may be used to evaluate the effectiveness of a regression test. Using this methodology, a software tester may evaluate just how effective a regression test is in identifying potential faults that were introduced in changes to the program..

The fifth milestone of this project comprised a presentation entitled, "A Detailed Look at the HAL/S Metric Analyzer." Subjects of the discussion included

- preparing an OI build for analysis
- preparing the HALMet analysis tool for operation
- using the HALMet analysis tool
- synopsis of principal components analysis
- preparing the PCA-RCM tool for operation
- using the PCA-RCM tool

The Measurement of Test Efficiency

The precise effects of changes to software modules can now be measured with the deltas in relative complexity. The magnitude of the deltas in ρ also show the likelihood of the introduction of software faults. That is, the larger the change in the value of the relative complexity of a module, the more likely that it is to have newly introduced faults. From a statistical testing perspective, test effort should be focused on those modules that are most likely to contain faults. Each program module that has been modified, then, should be tested in proportion the number of anticipated faults that might have been introduced into it.

Each program module is usually closely linked to a specific functionality. That is, as we exercise a particular functionality in PASS a particular execution profile emerges for that functionality as can be seen in Milestone V, Phase I. For each functionality, some modules have a high probability of being executed, while others have a low probability. These execution profiles were characterized by the probability distribution $P = \{p_i | 1 \leq i \leq n\}$.

The initial phase of the efficient testing of changed code is to identify the functionalities that will exercise the modules that have changed. Each of these functionalities so designated will have an associated test suite designed to exercise that functionality. With this information, we can now describe the efficiency of a test from a mathematical/statistical perspective.

Let $\Delta\rho_i^k = \left|\rho_i^k - \rho_i^{k-1}\right|$ represent the absolute change (delta) in relative complexity ρ_i^k of module *i* from release *k-1* to release *k*. For subsequent calculations, let $\delta_i^k = \Delta\rho_i^k$. Associated with any test *j* there will be an execution profile P^j for that test. The ***efficiency*** θ of the test *j* on release *k* is as follows:

$$\theta_j^k = \sum_i \delta_i^k p_i^j$$

This concept of test efficiency permits the numerical evaluation of a test on the actual changes that have been made to the software system. It is simply the expected value of the change in relative complexity from one release to another under a particular test. If the value of θ is large for a given test then the test will have exercised the changed modules. If the set of θ's for a given release is low, then it is reasonable to suppose that the changed modules have not been testing in proportion to the number of probable faults that were introduced during the maintenance changes.

An Analysis of the Test Results

Data prepared for this analysis included matched tests from OI23.10 and OI24.09. These data will be useful to demonstrate the proof of concept of regression test efficiency. Normally this analysis would be performed during the test process at all stages between successive or incremental builds. That is, we would normally compute the changes in relative complexity from two successive builds such as OI23.01 and OI23.02. The whole purpose of the computation of the deltas between these sequential builds is to guide the regression testing for the precise changes that have occurred between these two builds. The actual functional complexities that are obtained using the OI23.10 and OI24.09 data are quite small in that the tests for which PMIP data are available do not stress the execution of the changed modules as much as a specifically designed regression test would do.

To begin the measurement process between the two builds of OI23.10 and OI24.09 we must have a common baseline to compare the results of these two builds. The baseline that we have chosen to use is one that was established for the Milestone III deliverable. This baseline was obtained from OI23.01. We will use the baseline transformation matrix established in that study as a transformation vehicle in the current study so that the relative complexity values obtained in the current study may be related to previous deliverables. Again, the choice of the baseline transformation matrix is arbitrary. We could just as well have used a transformation matrix obtained from either OI23.10 or OI24.09.

The first step in the measurement process was to obtain the relative complexities for each of the high-level program modules for each of OI23.10 and OI24.09 systems. Examples of module relative complexities for each of the test program modules may be seen in Tables 10 and 11. These two tables list all program modules that have changed from OI23.10 to OI24.09. For each program module that has changed, the second column of the table shows the relative complexity value on OI23.10 and the third column shows the relative complexity for that same module on OI24.09. The fourth column, labeled Deltas, shows the

change (in absolute value) in relative complexity between OI23.10 and OI24.09. The final column shows the normalized signed percentage change from OI23.10 to OI24.09. The normalizing value for this computation was a relative complexity of 50. Thus, the adjusted percentage value for less complex modules are not dominated by relatively small changes to those modules.

Table 10. Deltas for All Changed Modules Ordered by Deltas

Module Name	OI23.10 Relative Complexity	OI24.09 Relative Complexity	Deltas	Percentage Change
GC1ORB (New Module)		48.59	48.59210	
GFDORB (New Module)		46.31	46.31330	
RCDDCO	60.09	53.23	6.86650	-13.73%
REXRMS	55.68	51.15	4.52859	-9.06%
RHMHLT	53.66	58.05	4.38365	8.77%
GTBUPL	65.31	68.99	3.68124	7.36%
GCQORB	67.34	70.95	3.60988	7.22%
GFKGRT	69.11	72.71	3.59790	7.20%
GFFORB	60.64	64.08	3.43670	6.87%
SSMANTMG	62.00	64.82	2.82124	5.64%
SM2OPS	67.62	69.84	2.21875	4.44%
SM4OPS	67.72	69.94	2.21875	4.44%
S4ICLN	45.84	43.65	2.18963	-4.38%
S2ICLNUP	45.76	43.59	2.17747	-4.35%
SSTHYDFL	54.20	55.84	1.64205	3.28%
SULUPLIN	77.89	79.45	1.56215	3.12%
RVMCON	58.26	59.36	1.09612	2.19%
DMPMMMSG	82.13	82.84	0.70931	1.42%
RFPPOS	43.90	44.54	0.64376	1.29%
ASLTMC	60.09	59.49	0.60365	-1.21%

Table 10. Deltas for All Changed Modules Ordered by Deltas

Module Name	OI23.10 Relative Complexity	OI24.09 Relative Complexity	Deltas	Percentage Change
PMQTEC	50.66	50.10	0.55701	-1.11%
GAALIM	49.66	49.11	0.55491	-1.11%
RDDDDI	48.27	48.81	0.53908	1.08%
GSCVEN	55.75	56.25	0.50540	1.01%
VS6SSTPR	69.22	68.80	0.41792	-0.84%
RSCSIN	45.80	45.38	0.41338	-0.83%
GO2ORB	79.98	80.26	0.27552	0.55%
PMRSLR	48.59	48.34	0.25088	-0.50%
GKWRMS	44.92	45.15	0.23031	0.46%
GRVLAN	48.44	48.22	0.22178	-0.44%
GR2ORB	49.17	49.38	0.20773	0.42%
GWKORB	54.77	54.59	0.18458	-0.37%
GR4ORB	47.46	47.63	0.17662	0.35%
GWDRMO	52.14	52.27	0.12848	0.26%
GO8ORB	69.42	69.54	0.12578	0.25%
GSQASC	59.55	59.67	0.12161	0.24%
PMWSLW	61.72	61.60	0.11984	-0.24%
VS5SSTPR	66.59	66.48	0.11039	-0.22%
GCUGCS	44.89	44.99	0.10064	0.20%
RPHCTF	53.42	53.52	0.09646	0.19%
GO3ENT	85.56	85.64	0.08295	0.17%
AIESIP	86.80	86.87	0.07066	0.14%
GKKORB	48.64	48.57	0.06896	-0.14%
DMIMCD	68.92	68.85	0.06855	-0.14%

Table 10. Deltas for All Changed Modules Ordered by Deltas

Module Name	OI23.10 Relative Complexity	OI24.09 Relative Complexity	Deltas	Percentage Change
GC9ORB	47.37	47.44	0.06713	0.13%
AIBGPCLO	92.27	92.33	0.06610	0.13%
DUPNSP	95.65	95.71	0.06160	0.12%
DMZLOG	43.58	43.52	0.05977	-0.12%
GEBADH	55.87	55.82	0.05043	-0.10%
RWPPHC	45.58	45.63	0.05034	0.10%
RBMHDW	44.08	44.12	0.04606	0.09%
VULMMINT	51.45	51.49	0.03883	0.08%
GECRDH	53.54	53.50	0.03580	-0.07%
DCD15301	42.96	43.00	0.03576	0.07%
GSTETS	65.26	65.22	0.03563	-0.07%
VM2BDYFL	58.51	58.54	0.03262	0.07%
GSMMPS	54.61	54.64	0.03005	0.06%
RXYCIN	48.08	48.10	0.02665	0.05%
ROVKYB	50.82	50.85	0.02638	0.05%
GACLIM	46.88	46.86	0.02277	-0.05%
GKMRMS	50.00	50.01	0.01492	0.03%
GC7ORB	49.74	49.76	0.01419	0.03%
GFATRA	49.32	49.34	0.01217	0.02%
SAMITEM	44.95	44.96	0.01087	0.02%
SSRREC	44.28	44.29	0.01087	0.02%
GFIGRT	53.32	53.31	0.01008	-0.02%
DCDDOW	54.80	54.81	0.00884	0.02%
SPCPPC	56.99	57.00	0.00884	0.02%

Table 10. Deltas for All Changed Modules Ordered by Deltas

Module Name	OI23.10 Relative Complexity	OI24.09 Relative Complexity	Deltas	Percentage Change
SSCFUELC	49.48	49.49	0.00884	0.02%
SSNO2N2Q	45.50	45.51	0.00884	0.02%
DCDDG3	46.01	46.02	0.00883	0.02%
SO2CONC	44.62	44.63	0.00883	0.02%
SPSPSP	74.42	74.43	0.00883	0.02%
SSHHYD	46.65	46.66	0.00883	0.02%
DCD14401	43.06	43.07	0.00825	0.02%
GH6ABR	48.43	48.44	0.00755	0.02%
ARAGPCSW	65.19	65.19	0.00681	0.01%
ARDCSBUS	63.48	63.48	0.00681	0.01%
SSAAPUFU	46.21	46.21	0.00681	0.01%
DCDDG2	46.08	46.09	0.00551	0.01%
GGTTAE	58.19	58.18	0.00401	-0.01%
GP2ORB	50.29	50.29	0.00187	0.00%
GMUHAN	51.60	51.60	0.00183	0.00%
GPW3AX	50.20	50.20	0.00064	0.00%
DCDDG1	46.10	46.10	0.00059	0.00%
GRTFIX	49.98	49.98	0.00021	0.00%
Net Change				+30.77%

Table 11. Deltas for All Changed Modules Ordered by Percentage Change

Module Name	OI23.10 Relative Complexity	OI24.09 Relative Complexity	Deltas	Percentage Change
GC1ORB (New Module)		48.59	48.59210	
GFDORB (New Module)		46.31	46.31330	
RHMHLT	53.66	58.05	4.38365	8.77%
GTBUPL	65.31	68.99	3.68124	7.36%
GCQORB	67.34	70.95	3.60988	7.22%
GFKGRT	69.11	72.71	3.59790	7.20%
GFFORB	60.64	64.08	3.43670	6.87%
SSMANTMG	62.00	64.82	2.82124	5.64%
SM2OPS	67.62	69.84	2.21875	4.44%
SM4OPS	67.72	69.94	2.21875	4.44%
SSTHYDFL	54.20	55.84	1.64205	3.28%
SULUPLIN	77.89	79.45	1.56215	3.12%
RVMCON	58.26	59.36	1.09612	2.19%
DMPMMMSG	82.13	82.84	0.70931	1.42%
RFPPOS	43.90	44.54	0.64376	1.29%
RDDDDI	48.27	48.81	0.53908	1.08%
GSCVEN	55.75	56.25	0.50540	1.01%
GO2ORB	79.98	80.26	0.27552	0.55%
GKWRMS	44.92	45.15	0.23031	0.46%
GR2ORB	49.17	49.38	0.20773	0.42%
GR4ORB	47.46	47.63	0.17662	0.35%
GWDRMO	52.14	52.27	0.12848	0.26%
GO8ORB	69.42	69.54	0.12578	0.25%
GSQASC	59.55	59.67	0.12161	0.24%

Table 11. Deltas for All Changed Modules Ordered by Percentage Change

Module Name	OI23.10 Relative Complexity	OI24.09 Relative Complexity	Deltas	Percentage Change
GCUGCS	44.89	44.99	0.10064	0.20%
RPHCTF	53.42	53.52	0.09646	0.19%
GO3ENT	85.56	85.64	0.08295	0.17%
AIESIP	86.80	86.87	0.07066	0.14%
GC9ORB	47.37	47.44	0.06713	0.13%
AIBGPCLO	92.27	92.33	0.06610	0.13%
DUPNSP	95.65	95.71	0.06160	0.12%
RWPPHC	45.58	45.63	0.05034	0.10%
RBMHDW	44.08	44.12	0.04606	0.09%
VULMMINT	51.45	51.49	0.03883	0.08%
DCD15301	42.96	43.00	0.03576	0.07%
VM2BDYFL	58.51	58.54	0.03262	0.07%
GSMMPS	54.61	54.64	0.03005	0.06%
RXYCIN	48.08	48.10	0.02665	0.05%
ROVKYB	50.82	50.85	0.02638	0.05%
GKMRMS	50.00	50.01	0.01492	0.03%
GC7ORB	49.74	49.76	0.01419	0.03%
GFATRA	49.32	49.34	0.01217	0.02%
SAMITEM	44.95	44.96	0.01087	0.02%
SSRREC	44.28	44.29	0.01087	0.02%
SSNO2N2Q	45.50	45.51	0.00884	0.02%
SSCFUELC	49.48	49.49	0.00884	0.02%
DCDDOW	54.80	54.81	0.00884	0.02%
SPCPPC	56.99	57.00	0.00884	0.02%

Table 11. Deltas for All Changed Modules Ordered by Percentage Change

Module Name	OI23.10 Relative Complexity	OI24.09 Relative Complexity	Deltas	Percentage Change
SO2CONC	44.62	44.63	0.00883	0.02%
DCDDG3	46.01	46.02	0.00883	0.02%
SSHHYD	46.65	46.66	0.00883	0.02%
SPSPSP	74.42	74.43	0.00883	0.02%
DCD14401	43.06	43.07	0.00825	0.02%
GH6ABR	48.43	48.44	0.00755	0.02%
SSAAPUFU	46.21	46.21	0.00681	0.01%
ARDCSBUS	63.48	63.48	0.00681	0.01%
ARAGPCSW	65.19	65.19	0.00681	0.01%
DCDDG2	46.08	46.09	0.00551	0.01%
GMUHAN	51.60	51.60	0.00183	0.00%
DCDDG1	46.10	46.10	0.00059	0.00%
GRTFIX	49.98	49.98	0.00021	0.00%
GPW3AX	50.20	50.20	0.00064	0.00%
GP2ORB	50.29	50.29	0.00187	0.00%
GGTTAE	58.19	58.18	0.00401	-0.01%
GFIGRT	53.32	53.31	0.01008	-0.02%
GACLIM	46.88	46.86	0.02277	-0.05%
GSTETS	65.26	65.22	0.03563	-0.07%
GECRDH	53.54	53.50	0.03580	-0.07%
GEBADH	55.87	55.82	0.05043	-0.10%
DMZLOG	43.58	43.52	0.05977	-0.12%
DMIMCD	68.92	68.85	0.06855	-0.14%
GKKORB	48.64	48.57	0.06896	-0.14%

Table 11. Deltas for All Changed Modules Ordered by Percentage Change

Module Name	OI23.10 Relative Complexity	OI24.09 Relative Complexity	Deltas	Percentage Change
VS5SSTPR	66.59	66.48	0.11039	-0.22%
PMWSLW	61.72	61.60	0.11984	-0.24%
GWKORB	54.77	54.59	0.18458	-0.37%
GRVLAN	48.44	48.22	0.22178	-0.44%
PMRSLR	48.59	48.34	0.25088	-0.50%
RSCSIN	45.80	45.38	0.41338	-0.83%
VS6SSTPR	69.22	68.80	0.41792	-0.84%
GAALIM	49.66	49.11	0.55491	-1.11%
PMQTEC	50.66	50.10	0.55701	-1.11%
ASLTMC	60.09	59.49	0.60365	-1.21%
S2ICLNUP	45.76	43.59	2.17747	-4.35%
S4ICLN	45.84	43.65	2.18963	-4.38%
REXRMS	55.68	51.15	4.52859	-9.06%
RCDDCO	60.09	53.23	6.86650	-13.73%

Tables 10 and 11 contain exactly the same data. The data in Table 10 are ordered by the absolute change (Deltas) made to individual program modules. The data in Table 11 are ordered by the normalized percentage change. For both Table 10 and Table 11, there are two modules that stand out: the first two modules in each of these tables. These two modules were added in the builds between OI23.10 and OI24.09. The net change in normalized percentage between OI23.10 and OI24.09 was 30.77%. The precise nature of the changes that have occurred may be seen in the Section III report.

From the standpoint of changes in relative complexity, the first 20 program modules in Table 10 account for the majority of total change to system complexity. In accordance with the relationship between the fault surrogate, relative complexity, and faults, it would be reasonable to suppose that these modules are the ones into which new faults had the greatest likelihood of being introduced. These, then, are the modules that should receive the greatest attention in subsequent testing effort.

The PMIP processor will permit us to estimate the total time spent in each program module while the PMIP processor is running. From these time estimates, we may then construct the *execution profile* for each test. We have data for seven such tests that were run on both OI23.10 and OI24.09. These execution profiles and their associated tests are presented in Tables 12-18. The column labeled Execution Profile in each of these tables represents the proportion that each module received during the execution of the test represented by the table. The values in the execution profile column do not add up to 100%. We have tabled only those profile values for those program modules that have non-zero delta values from Table 10.

Again, the test data we were working with did not represent actual regression tests of the code that had changed but were used for the purposes of discussion. Had these tests actually been constructed to test for the changes that had been made, the tests would have been demonstrably inadequate. In each of the Tables 12-18, the modules that were in our top 20 list for changes are marked with asterisks. In Table 12, only four of these modules made the test. In Tables 13, 14, 15, 17, and 18, none of the modules in the top 20 list are represented here. Only the tests represented by Tables 12, 16, and 18 had any modules from the top 20 list. Clearly, these sample tests are seen to be inadequate based on their inability to execute the modules in which most of the changes had occurred.

In each of the Tables 12-18 there are three numerical data columns for each module. The contents of the first column is execution profile p_i^j for the i^{th} module on the j^{th} test. The second column is the absolute change in relative complexity δ_i of the i^{th} module from OI23.10 to OI24.09. The third column in each of the tables, labeled Test Efficiency, is the product of the previous two. From the standpoint of the formal definition of test efficiency, the entries in this column represent the test efficiencies θ_{ij} for each program module $\theta_{ij} = \delta_i p_i^j$ where *j* is the number of the test.

The maximum possible value of test efficiency is 6.87 if we discount the fact that two new modules were added whose delta values are the modules themselves. If each test were to assign all of its activity to the module, **RCDDCO**, that had the highest change, then we would spend 100% of our time executing this module to the exclusion of all others. Hence, the upper bound on test efficiency is equal to the value of the largest delta. This is not necessarily the best test, however. In the trade-off between optimality and fairness, all modules should receive test effort in proportion to the degree of change within the module. This being the case, the maximum fair value for test efficiency would be 1.173 (again, discounting the new modules).

At the bottom of each of the Tables 12-18 there are two values recorded. The first of these is labeled Total. This value represents the sum of the test efficiency column. It represents the test efficiency of each of the tests for changes that have been made to all program modules. In order to scale these number so that they might be more readily interpreted, the last row in each table shows the test efficiency as a percentage of the maximum possible value of 6.87. As we look at each table, we can see that the maximum test efficiency was achieved by the test reported in Table 16. In this case the value is close to 5% of the

maximum. Though this is the largest of all of the test efficiencies, it is still not very impressive if we were to attempt to assert that this test was a good one. At the other extreme, at the low end, is the test represented by the data in Table 14. Here the test efficiency is the lowest at 0.04%.

Conclusions and Recommendations

The concept behind the computation of test efficiency is that it is possible to evaluate or measure test outcomes in terms of the potential each test has for exposing the potential faults that have been introduced through changes to the code. As this current project has progressed, we have gradually built a reasonable surrogate for software faults. This surrogate is relative complexity. We have systematically selected for incorporation into the computation of relative complexity those measures that varied directly with software faults. Each of the selected measures explained a significant proportion of variation in the corresponding measure of software faults. In this context, the larger the value of relative complexity, the greater the number of explained or potential faults that a module might have.

As a direct result of an effective test process, the faults from a particular set of code may be found and removed. Ultimately, it is conceivable that a large number of the residual faults in a program may be removed. Problems arise when this code is disturbed as additions and deletions are made to the code. The purpose of the current investigation was to develop and demonstrate a measure that could be applied to the regression testing process that would evaluate the overall effectiveness of this process. Thus the notion of test efficiency was developed.

This current study can only demonstrate the possibilities for the use of this new measure. For test efficiency to be fully validated, the data must be obtained directly from the test process at its inception at each new build of PASS. It would be a worthwhile investigation for the PMIP processor to be enabled for future testing of new builds for PASS. These data, in conjunction with the existing relative complexity, should be employed as evaluative criteria for the effectiveness of the testing process. In addition, changes that are made to individual program modules as a result of faults uncovered during the test and inspection process should be noted for the full validation of the test efficiency concept. This study represents the first step in the process of the measurement of test efficiency for changed code.

Table 12. Tests ORBCK23Z-249

Module	Execution Profile	Deltas	Test Efficiency
GFFORB*	0.06914952	3.43670	0.23764616
GCQORB *	0.03118250	3.60988	0.11256508
GR2ORB	0.01376751	0.20773	0.00285992
GAALIM	0.00453390	0.55491	0.00251591
GR4ORB	0.01304194	0.17662	0.00230347
AIESIP	0.03074037	0.07066	0.00217211
ASLTMC *	0.00140665	0.60365	0.00084912
GWKORB	0.00298145	0.18458	0.00055032
DMIMCD	0.00659055	0.06855	0.00045178
DCDDOW	0.02600489	0.00884	0.00022988
GC9ORB	0.00299455	0.06713	0.00020102
GC7ORB	0.01054558	0.01419	0.00014964
DCDDG2	0.01655312	0.00551	9.1208E-05
GP2ORB	0.02712692	0.00187	5.0727E-05
ARAGPC	0.00260071	0.00681	1.7711E-05
GTBUPL *	0.00000166	3.68124	6.1109E-06
DMZLOG	0.00006493	0.05977	3.8809E-06
GO2ORB	0.00000870	0.27552	2.3970E-06
GRTFIX	0.00417561	0.00021	8.7688E-07
DUPNSP	0.00000992	0.06160	6.1107E-07
GKKORB	0.00000227	0.06896	1.5654E-07
Total			0.3626681
% of Maximum			5.28%

Table 13. Tests DEOCK23Z-249

Module	Execution Profile	Deltas	Test Efficiency
AIESIP	0.03759479	0.07066	0.00265645
GSCVEN	0.00130921	0.50540	0.00066167
DMIMCD	0.00814676	0.06855	0.00055846
DCDDOW	0.03226015	0.00884	0.00028518
DCDDG3	0.02656619	0.00883	0.00023458
GACLIM	0.00191361	0.02277	4.3573E-05
ARAGPC	0.00299542	0.00681	2.0399E-05
GPW3AX	0.01719282	0.00064	1.1003E-05
GRTFIX	0.00985202	0.00021	2.0689E-06
DMZLOG	0.00001143	0.05977	6.8317E-07
DUPNSP	0.00001048	0.06160	6.4557E-07
GO3ENT	0.00000231	0.08295	1.9161E-07
Total			0.00447491
% of Maximum			0.07%

Table 14. Tests ENOMK23Z-249

Module	Execution Profile	Deltas	Test Efficiency
AIESIP	0.02399549	0.07066	0.00169552
GSCVEN	0.00101414	0.50540	0.00051255
DMIMCD	0.00540574	0.06855	0.00037056
DCDDOW	0.02119384	0.00884	0.00018735
DCDDG3	0.01741212	0.00883	0.00015375
GRVLAN	0.00035325	0.22178	7.8344E-05
GACLIM	0.00123505	0.02277	2.8122E-05
ARAGPC	0.00198728	0.00681	1.3533E-05
GGTTAE	0.00103280	0.00401	4.1415E-06
GRTFIX	0.00674501	0.00021	1.4165E-06
DMZLOG	0.00000633	0.05977	3.7834E-07
GO3ENT	0.00000157	0.08295	1.3023E-07
GPW3AX	0.00016687	0.00064	1.0680E-07
Total			0.00304591
% of Maximum			0.04%

Table 15. Tests RTLSK23Z-249

Module	Execution Profile	Deltas	Test Efficiency
GFKGRT	0.00339348	3.59790	0.01220940
AIESIP	0.02507115	0.07066	0.00177153
GCUGCS	0.01017962	0.10064	0.00102448
DMIMCD	0.00564324	0.06855	0.00038684
DCDDOW	0.02234324	0.00884	0.00019751
GSQASC	0.00145211	0.12161	0.00017659
GFIGRT	0.01724639	0.01008	0.00017384
GRVLAN	0.00060135	0.22178	0.00013337
GACLIM	0.00112650	0.02277	2.5650E-05
ARAGPC	0.00207811	0.00681	1.4152E-05
GSMMPS	0.00040180	0.03005	1.2074E-05
DCDDG1	0.01689900	0.00059	9.9704E-06
GSTETS	0.00024204	0.03563	8.6239E-06
GRTFIX	0.00549054	0.00021	1.1530E-06
DMZLOG	0.00000819	0.05977	4.8952E-07
GECRDH	0.00000595	0.03580	2.1301E-07
GPW3AX	0.00019781	0.00064	1.2660E-07
GEBADH	0.00000234	0.05043	1.1801E-07
Total			0.01614614
% of Maximum			0.24%

Table 16. Tests ORB2K23Z-249

Module	Execution Profile	Deltas	Test Efficiency
GFFORB *	0.06390412	3.43670	0.21961929
GCQORB *	0.03615104	3.60988	0.13050092
GR2ORB	0.01594873	0.20773	0.00331303
GAALIM	0.00513233	0.55491	0.00284798
GR4ORB	0.01510796	0.17662	0.00266837
AIESIP	0.03611892	0.07066	0.00255216
ASLTMC *	0.00163716	0.60365	0.00098827
GWKORB	0.00348506	0.18458	0.00064327
DMIMCD	0.00769291	0.06855	0.00052735
DCDDOW	0.03013646	0.00884	0.00026641
GC7ORB	0.01247349	0.01419	0.00017700
GP2ORB	0.03122655	0.00187	5.8394E-05
ARAGPC	0.00301628	0.00681	2.0541E-05
GC9ORB	0.00029623	0.06713	1.9886E-05
DMZLOG	0.00018258	0.05977	1.0913E-05
GO8ORB	0.00002257	0.12578	2.8389E-06
GRTFIX	0.00484038	0.00021	1.0165E-06
Total			0.36421763
% of Maximum			5.30%

Table 17. Tests VUG9K23Z-249

Module	Execution Profile	Deltas	Test Efficiency
AIESIP	0.08062626	0.07066	0.00569705
DMIMCD	0.01840358	0.06855	0.00126157
DCDDOW	0.07095703	0.00884	0.00062726
ARAGPC	0.00707823	0.00681	4.8203E-05
DMZLOG	0.00039032	0.05977	2.3329E-05
Total			0.00765741
% of Maximum			0.11%

Table 18. Tests VUP9K23Z-249

Test #8 (P9)			
Module	**Execution Profile**	**Deltas**	**Test Efficiency**
DMPMMM	0.05050691	0.70931	0.03582506
AIESIP	0.15213673	0.07066	0.01074998
ASLTMC *	0.00770360	0.60365	0.00465028
DMIMCD	0.03464441	0.06855	0.00237487
DCDDOW	0.13331710	0.00884	0.00117852
ARAGPC	0.01323108	0.00681	9.0104E-05
DMZLOG	0.00067702	0.05977	4.0465E-05
Total			0.05490928
% of Maximum			0.80%

Section V
Severity 1 Software Faults Analysis

Introduction

In an attempt to characterize the internal nature of computer programs, many measures of software attributes have been developed and investigated. These software metrics represent quantitative descriptions of program attributes. Some of these metrics have been shown to be related, somehow, to quantitative measures of program quality. Software metrics can be obtained relatively early in the software life-cycle. Measures of software quality, on the other hand, are generally developed over a longer time. A program of any size must be run several years for the latent faults in it to be found. Thus, if we were to use the number of program faults as a software quality measure, this information could probably be obtained only after the program had reached obsolescence. Historical data from past program development and maintenance scenarios may be used to develop predictive models. In this case, metric data that were obtained early in the development process can be incorporated into a predictive model with a criterion variable related to a measure of software quality obtained over time.

The foundations for the predictive model development based on software complexity metrics are clear. There is a relationship between measures of software complexity and errors during the program development and operational phases. However, there are no viable models at present that reflect program complexity and potentials for program complexity in the prediction of error phenomena. There is a clear intuitive basis for believing that complex programs have more errors in them than simple programs. There is reasonable evidence to support the conclusion that computer software complexity metric models may be integrated with software quality models.

By way of clarification of the intent of this study, it will be necessary to distinguish between the two terms, error and fault. An error is a human action that results in a software product that contains a fault. As a consequence of one or more failures of a program (i.e., events in which a system or system component does not perform a required function within specific limits), a fault will be discovered and a change made to the program segment.

The software complexity metrics now collected by the HAL/S metric analyzer, HALMet, have been shown to be closely related to the distribution of faults in program modules. Many researchers have shown that it is possible to develop a predictive relationship between complexity metrics and faults. We are interested, then, in investigating the relationship between the numbers of faults (changes) in programs and the particular software metrics that may be used to classify programs to groups of similar fault (change) characteristics. In particular, we are most interested in the ability of the metrics to predict the occurence of particular types of faults in software modules. Not all faults are of equal interest to us in this investigation. Software faults in the Space Shuttle PASS environment are classified into three severity levels. A Severity 1 fault will result in the loss of the shuttle and its crew. Its effects are potentially catastrophic. A Severity 2 fault will likely result in an abort condition and cause a flight to be scrubbed. A Severity 3 fault may, in fact, go totally unnoticed to the flight crew or ground controllers.

The state of the art in the use of complexity metrics as leading indicators of program quality is not good. In current practice, many complexity metrics have not been validated; we are not sure just what they are measuring. Even the collection of trouble reports for program is subject to great variability, even within a single development group. We have investigated the potential use of multivariate regression techniques to use complexity metrics as predictors of program quality. The results of this work suggest that metrics do have great potential as predictors of program quality. However, in the short term, we believe that other predictive techniques might yield better results than the direct prediction of measures of program quality. A major problem in the application of multiple regression techniques for the prediction of numbers of faults in programs, for example, is that the distribution of faults is heavily skewed in favor of programs that have no faults or a small number of faults. This is so because the majority of program modules in a software system will typically exhibit either no faults or very few faults during the latter stages of the software life-cycle.

A predictive technique that does show promise for use in the circumstance of noisy and certainly non-normally distributed data is that of discriminant analysis. In contrast to regression modeling techniques, discriminant analysis will not be used to estimate the number of potential faults in program module, a somewhat tenuous proposition given the nature of the current data collection processes. Rather, discriminant analysis will be used to assign program modules to groups of modules of similar characteristics. In this investigation we will use this technique to develop assignment models that will use complexity information provided by the HALMet tool to assign programs into one of two mutually exclusive groups. The model for the first group will be derived from the complexity characteristics of programs that have been found to contain no faults. An alternate model will be developed for programs observed to have had at least one Severity 1 fault. The overall objective of this procedure is to use past development metric and quality data to develop a model and then use this model to assign new program modules under development and those that have received major changes to whichever group its metric profile most closely matches. From this basic model we could then identify a program or a program

module that had a complexity profile that would cause it to be associated with a high-severity fault group. Similarly, the model might assign a module to a group characterized by no faults. The specific utility of this approach is that it permits limited testing resources to be focused on the modules whose complexity characteristics have led to problems in past development work.

The research effort for this deliverable focused on the feasibility of the construction of a statistical software filter to aid in the identification of those regions of the onboard flight software that are most prone to Severity 1 failures. In this phase, the software faults (DRs) will be classified as to their severity level 1, 2, or 3. A discriminant model based on orthogonal software metrics was employed to test the hypothesis as to whether it is possible to uniquely characterize the software attributes that are associated with software faults of Severity 1. If it is possible to identify such attributes, then a software tool will be implemented for the HAL/S language using the discriminant function to examine existing software modules and future software modules for those regions that are most likely to contain Severity 1 faults. The prime deliverable at this stage (Milestone VI) is the report documenting the statistical analysis.

Modeling Methodology

The use of the discriminant analysis technique in software engineering is certainly not new or novel. It is our intention to explore the viability of the technique as a tool for classifying program modules as to their potential quality based on the complexity measures of the programs. What is novel in our approach to the problem is the fact that the discriminant model will contain a deliberate distortion to classify program modules into extreme groups: those containing few faults and those containing many faults.

The statistical technique of discriminant analysis is basically a classification procedure. The underlying principle of the technique is that an operational hypothesis is formulated that there exists an a priori classification of multivariate observations into two or more groups or sets of observations. Further, the membership in one of these supposed groups is mutually exclusive. A criterion variable will be used for this group assignment. Thus, a program module, for example, might be classified with a code of 1 if it has been found to have one or more Severity 1 faults or with a code of 2 if it has no fault recorded.

In the application of discriminant analysis in this study, uncorrelated measures of program complexity (domain metrics) will be used as independent variables in an attempt to classify programs into a group whose programs contain relatively few faults or to a group whose programs contain a relatively large number of faults. These metric variables will serve as discriminating variables that measure the characteristics on which the two groups of programs (those with faults and those without) are expected to differ. The traditional discriminant analysis proceeds by forming (for the two-group problem) two linear combinations of discriminating variables. These discriminant functions are employed to produce discriminant scores for individual observations. The presumption here is that the scores within a particular group will be fairly similar.

A general problem arises in the use of complexity metric data in statistical procedures. Many of the underlying statistical assumptions cannot be met. To construct a simple linear discriminant function there must be homogeneity of the *within* groups covariance matrices. That is, the assumption is made for this type of analysis that all observations represent a sample drawn from the same population. When complexity metric data are to be employed as predictors of quality, this criterion seldom can be met. In the presence of heterogeneous *within* groups covariance matrices, where it is obvious that the measures in the several groups are drawn from differing distributions, an alternate procedure must be used. In this case the discriminant technique called logistic discrimination will be used. This technique will compute the posterior probabilities of group membership from the prior probabilities of group membership, the group means, and a pooled covariance matrix.

In the case of heterogeneous *within* group covariance matrices, the linear discriminant function is not constructed. Rather, an estimate of the posterior probability of group membership is constructed using the logistic discriminant procedure. For the two-group case, the posterior probability of an observation **x** belonging to one of two groups is given by

$$p_j \boldsymbol{x} = \frac{e^{-\frac{1}{2} D_j^2 \boldsymbol{x}}}{\sum_i e^{-\frac{1}{2} D_i^2 \boldsymbol{x}}}$$

where

$$D_j^2 \boldsymbol{x} = (\boldsymbol{x} - \bar{\boldsymbol{x}}_j)^T \boldsymbol{\Sigma} (\boldsymbol{x} - \bar{\boldsymbol{x}}_j)$$

is the generalized squared distance from the observation represented by the vector $\boldsymbol{x}$ to the group *j* represented by its mean, $\bar{\boldsymbol{x}}_j$, and $\boldsymbol{\Sigma}$ is the pooled covariance matrix. In the case of two groups the values of *i* and *j* are simply 1 and 2.

The focus of this investigation is on the application of discriminant analysis to explore the nature of the relationship between measures of software complexity and those of software quality. The ability to develop predictive models for this relationship is desirable for a number of reasons. Given the viability of the classifications models and the fact that the measures of program complexity may be obtained early in the software development process, program modules most likely to contain faults may be identified as they are prepared. To the extent that a definitive relationship between the software metrics and an assessment of known faults can be established, these metrics will serve as leading indicators of program reliability. Also, program modules that will later prove difficult to maintain may also be identified.

The Discriminant Analysis of PASS

The overall objective of this study is to explore the possibilities for developing a useful discriminant procedure that will permit the identification of those program modules most prone to faults or changes. As was mentioned earlier, the present state of the art in software measurement is not very precise. As a result it is probably not possible to develop precise predictive models using the usual regression techniques. As an intermediate step in the refinement of predictive techniques, however, it would be useful if a model could be developed that would serve to identify those program modules most likely to require changes. If this were possible, the limited resources of the testing and maintenance processes could be focused on the modules most likely to cause problems. Discriminant analysis may be used effectively to solve this type problem.

There are two distinct aspects of the creation of a viable discriminant analysis model. First, the discriminant model must be able to perform the classification function with relatively little ambiguity. Second, the model must be able to classify future observations equally successfully. To this end, the 572 high-level program modules were split into three groups. The first group of 23 program modules contained those that had demonstrated at least one Severity 1 fault. The second group were those containing faults of Severity 2 and Severity 3. The third group was one in which no DRs were recorded. Normally, the criterion variable, DRs, would be used to assign the observations to two or three mutually exclusive groups. In this case, three distinct groups of programs were established based on the DRs.

At this point, the members of Group 2 were discarded, and were not used in the development of the discriminant model. The discriminant analysis was performed only on Groups 1 and 3. In that the remaining modules in Group 3 contained 190 program modules, these data would clearly dominate any discriminant function over the 23 observations in Group 1. Thus, a random sample of 67 program modules were drawn from the original set for a total of 90 observations for the discriminant model. This, of course, created a deliberate distortion in the data and the subsequent model that was developed using these data. The objective was to create a model that would, in a sense, magnify the differences between the set of program modules whose characteristics resulted in Severity 1 faults and the characteristics of the programs that had none. It is the stated purpose of this process to develop a methodology that will serve as a filter for the identification of the program modules most likely to cause Severity 1 failures after they are placed in service.

The basic discriminant analysis of variance that resulted is shown below in Table 19. All of the set of 18 metrics is seen to contribute to the formulation of the resulting model. The Wilks' Lambda for this analysis was 0.262, which was significant ($Pr < 0.01$). This means that the resulting model was able to discriminate between the set of modules in Group 1 from those in Group 3. Another way of stating this is that the centroids of the two groups are distinctly separated. This means that the set of working metrics that we

have identified for our measurement purposes are tools that may be used to identify models that also have the potential to contain Severity 1 problems.

Table 19. Discriminant ANOVA

Variable	STD	Total STD	Pooled STD	Between R-Squared	Pr > F
Eta1	10.12	8.75	7.27	0.260	0.0001
Eta2	95.87	70.99	91.24	0.457	0.0001
N1	749.83	567.69	694.09	0.433	0.0001
N2	375.82	287.81	342.56	0.420	0.0001
Stmts	181.01	129.38	179.06	0.494	0.0001
Loc	283.60	202.24	281.22	0.497	0.0001
Comments	231.57	147.10	252.47	0.601	0.0001
Nodes	84.98	62.77	81.11	0.460	0.0001
Edges	118.77	88.12	112.75	0.455	0.0001
Paths	19426.00	15265.00	17048.00	0.389	0.0001
Cycles	5.72	5.53	2.23	0.077	0.0081
Maxp	82.07	69.92	61.33	0.282	0.0001
Avgp	79.36	68.83	56.49	0.256	0.0001
Ds	73.38	62.32	55.26	0.286	0.0001
Sets	0.98	0.94	0.42	0.095	0.0031
Resets	1.10	1.09	0.29	0.035	0.0737
Maxsear	0.26	0.23	0.18	0.245	0.0001
Maxrear	0.20	0.20	0.07	0.059	0.0202

The basic discriminant function that emerged from this analysis is presented in Table 20. There are two coefficient structures in this table. The first column of the table lists each of the metric variables in the model. The second column is the Class 1 column. It contains the coefficients for the assignment probability to the first group. This is the group containing as its members those program modules that

have had Severity 1 DRs recorded against them. The second data column labeled Class 2 will assign to each observation the probability of group membership to the class of program modules that do not contain any DRs against them.

Table 20. The Discriminant Model

Variable	CLASS	
	1	2
Constant	-12.05718	-3.23276
Eta1	0.35150	0.43050
Eta2	-0.00244	-0.02333
N1	-0.04590	-0.00355
N2	0.05778	0.00381
Stmts	0.04740	-0.00404
Loc	0.01231	0.00642
Comments	0.02016	-0.00144
Nodes	0.09366	0.17984
Edges	0.05824	-0.09041
Paths	-0.00013	-0.00003
Cycles	1.06585	0.49148
Maxp	0.22921	-0.17148
Avgp	-0.49036	0.10130
Ds	0.01167	0.00678
Sets	6.01484	-0.17031
Resets	-4.96337	-0.58320
Maxsear	-3.29532	1.32406
Maxrear	10.62562	-1.04622

The purpose of discriminant analysis, at this stage, is to assign a probability of group membership to each of the program modules. The program module will then be assigned to the group for which it has the

greatest probability of membership. For the purposes of presentation in these tables, there are only two groups, labeled as Class 1 and Class 2. Class 1 is the group with Severity 1 DRs, whereas Class 2 is the group with no DRs. The results of the assignment of the discriminant model to classes is shown in Table 21 below.

To analyze the a posteriori performance of the discriminant model let us assume that one of two distinct faults of misclassification might occur. A Type I error will be the case where we conclude that a program is fraught with faults when in fact it is not. A Type II error will be the case where we believe that a program is relatively fault-free when in fact it is not. Necessarily, it is better to make a Type I error of misclassification than it is to make a Type II error.

Table 21. Number of Observations and Percent Classified Into CLASS

From CLASS	1	2	Total
1	18	5	23
	78.26	21.74	100.00
2	1	66	67
	1.49	98.51	100.00
Total	19	71	90
Percent	21.11	78.89	100.00

There are several ways that the results of the discriminant analysis may be examined to determine the relative rates of Type I and II errors. First, let us examine the rate of misclassification within the two groups of validation data where the first group consists of those program modules actually containing Severity 1 faults (Class 1) and the second group consisting of the program modules containing no faults (Class 2). The Class 1 data are shown in the second column of Table 21. In this group there are 1 out of 67 cases of Type I errors for a net misclassification rate of about 1%. The Class 2 data are shown in the third column of this table. Here there are 5 misclassifications out of 23 for a net Type II error rate of about 20%. The two cases considered here contain only the validation program data for those values of the criterion variable that were used to develop the discriminant model.

The posterior probability for group membership for each of the 90 test cases is shown in Table 22. The first column of this table is the observation number representing the particular program module. The second column is the actual class to which the module in question belonged. Class 1 modules were those that contained at least one Severity 1 fault. Class 2 modules were those that had no DRs filed against them. The next column shows how the modules were classified by the discriminant function. Those marked with asterisks are those that were misclassified. The fourth column shows the probability of membership of each module in Class 1. The fifth column shows the probability of membership in Class 2.

What is striking about the last two columns of the data presented in the table are the extremely high classification probabilities. The resulting model is very definitely polarizing the two sets of program modules with a high probability. The next logical step in the refinement of this modeling process is to identify exactly what were the circumstances that led modules such as numbers 1 and 2 to be misclassified with almost total certainty. A detailed analysis of such modules would almost certainly identify one or more measurable attributes directly related to the Severity 1 faults that we are not now measuring.

Table 22. Posterior Probability of Membership in Class

Obs	From CLASS	Classified Into CLASS	1	2
1	1	2*	0.0000	1.0000
2	1	2*	0.0000	1.0000
3	1	1	1.0000	0.0000
4	1	2*	0.0151	0.9849
5	1	1	1.0000	0.0000
6	1	1	1.0000	0.0000
7	1	2*	0.1952	0.8048
8	1	1	1.0000	0.0000
9	1	2*	0.0287	0.9713
10	1	2*	0.0000	1.0000
11	1	1	0.9287	0.0713
12	1	1	0.9999	0.0001
13	1	1	0.9994	0.0006
14	1	2*	0.0510	0.9490
15	1	1	1.0000	0.0000
16	1	2*	0.0218	0.9782
17	1	2*	0.0230	0.9770
18	1	1	0.9939	0.0061

Table 22. Posterior Probability of Membership in Class

Obs	From CLASS	Classified Into CLASS	1	2
19	1	1	1.0000	0.0000
20	1	1	0.9913	0.0087
21	1	2*	0.3049	0.6951
22	1	1	1.0000	0.0000
23	1	2*	0.0035	0.9965
24	2	2	0.0022	0.9978
25	2	2	0.0006	0.9994
26	2	2	0.0001	0.9999
27	2	2	0.0001	0.9999
28	2	2	0.0062	0.9938
29	2	1*	1.0000	0.0000
30	2	2	0.0001	0.9999
31	2	2	0.0001	0.9999
32	2	2	0.0008	0.9992
33	2	2	0.0000	1.0000
34	2	2	0.0014	0.9986
35	2	2	0.3829	0.6171
36	2	2	0.1056	0.8944
37	2	2	0.0030	0.9970
38	2	2	0.0003	0.9997
39	2	2	0.2368	0.7632
40	2	2	0.4027	0.5973
41	2	2	0.0024	0.9976
42	2	2	0.0017	0.9983
43	2	2	0.2315	0.7685
44	2	2	0.0001	0.9999
45	2	1*	0.9897	0.0103
46	2	2	0.0001	0.9999
47	2	2	0.0001	0.9999
48	2	2	0.0004	0.9996
49	2	2	0.0279	0.9721
50	2	2	0.0288	0.9712
51	2	2	0.0007	0.9993

Table 22. Posterior Probability of Membership in Class

Obs	From CLASS	Classified Into CLASS	1	2
52	2	2	0.0004	0.9996
53	2	2	0.0003	0.9997
54	2	2	0.0374	0.9626
55	2	2	0.3386	0.6614
56	2	2	0.0002	0.9998
57	2	2	0.0003	0.9997
58	2	2	0.0003	0.9997
59	2	2	0.0058	0.9942
60	2	2	0.0001	0.9999
61	2	2	0.0002	0.9998
62	2	2	0.0010	0.9990
63	2	2	0.0029	0.9971
64	2	2	0.0001	0.9999
65	2	2	0.0001	0.9999
66	2	2	0.0003	0.9997
67	2	2	0.0005	0.9995
68	2	2	0.0004	0.9996
69	2	2	0.0007	0.9993
70	2	2	0.0008	0.9992
71	2	1*	0.8352	0.1648
72	2	2	0.0000	1.0000
73	2	2	0.0010	0.9990
74	2	2	0.0011	0.9989
75	2	2	0.0002	0.9998
76	2	2	0.0000	1.0000
77	2	2	0.0373	0.9627
78	2	2	0.0018	0.9982
79	2	2	0.0001	0.9999
80	2	2	0.0001	0.9999
81	2	2	0.0002	0.9998
82	2	2	0.0006	0.9994
83	2	2	0.0001	0.9999
84	2	2	0.0007	0.9993

Table 22. Posterior Probability of Membership in Class

Obs	From CLASS	Classified Into CLASS	1	2
85	2	2	0.0001	0.9999
86	2	2	0.0001	0.9999
87	2	2	0.0000	1.0000
88	2	2	0.0084	0.9916
89	2	2	0.0000	1.0000
90	2	2	0.0006	0.9994

* Misclassified observation

The bottom line of this particular discriminant procedure is that the procedure was able to make a significant classification model for these data. The data were remarkably thin. We were able to identify only 23 of the modules that had DRs that were classified as Severity 1. There were certainly many more of these modules that had this property that were modified before the current DR tracking system was first employed. These modules would be misclassified by us in this analysis as Class 2 modules when in fact they were Class 1 modules. Nonetheless, the strength of the association between the things that we are measuring and the Severity 1 fault class is very great. This would indicate very good potential for the future use of this technique to aid in the identification of modules containing potential catastrophic faults.

Summary

The specific focus of this study has been to investigate some aspects of the relationship between program complexity measures and program that may lead to catastrophic software problems in PASS. The particular statistical vehicle chosen to measure this relationship was that of discriminant analysis. Based on the experimental observations contained herein, we believe there is a useful relationship between program faults and the complexity domains that HALMet now measures. Further, the strength of this relationship suggests that predictive models are indeed possible for the determination of program faults complexity domains.

The basic technique we have developed in this study relates to some major problems we have observed in the effort to develop reliable and meaningful predictors for program modules in terms of the number of faults that these modules might contain. Software complexity metrics certainly would be useful in this regard in that they are numerical measures which may be obtained before the test and validation of a program. The present state of the art in terms of the prediction of software quality in practical applications is not good. Most preliminary assessments of software quality are performed in an ad hoc manner by

experienced software engineering managers. The discriminant analysis procedure presented here represents a crucial step in the direction of a more rigorous approach to the prediction of software quality. One of the principal virtues of this methodology is that historical data from past development projects may be used as a baseline to build a statistical model for the prediction of quality measures of similar software systems under development. As time progresses, a metric database may be developed and augmented with the results of each new software product as it is developed and deployed.

As our understanding of the relationship between complexity metrics and software quality measures becomes more clear, the implications for software development practice are profound. Those programs with the complexity characteristics that will lead to a deterioration of software quality may be identified at an early stage. If nothing else, the limited resources of the testing process may be focused on the programs most likely to cause problems. There is also great potential for these data to feed back to the design stage. If, for example, certain design practices lead to programs whose complexity characteristic will create software quality problems, these design practices may be systematically modified. A lingering concern of ours is the current receptivity within the software engineering community for qualitative as opposed to quantitative software design and development models.

Section VI
Software Reliability Assessment Tool Set

Introduction

The specific deliverables for Milestone 7 included:

1. The software system. The Dynamic Reliability Assessment Tool (DRAT) is implemented in the C programming language and will be designed to run in the standard UNIX workstation environment. The source code is included.
2. Program reference manual. This deliverable will document how the software will be used on the Unix workstation.
3. User handbook. This deliverable will assist the user in the preparation of the data for input to the system. The specific data inputs will consist of system performance data as obtained from a system profiler such as PMIP and fault data for each of the program modules in the system.

The Theoretical Foundation for Dynamic Reliability Assessment

The traditional approach to the modeling of software reliability is based on a philosophical approach that began with attempts to model hardware reliability. Inherent in this approach is the concept of the failure event and the fact that it is possible to identify with some precision this failure event and measure the elapsed time to the failure event. For hardware systems this has real meaning. Take, for example, the failure of a light bulb. A set of light bulbs can be switched on and a very precise timer started for the time that they were turned on. One by one the light bulbs will burn out and we can note the exact time to failure of each of the bulbs. From these failure data we can then develop a precise estimate for both the mean time to failure for these light bulbs and a good estimate of the variance of the time to failure.

The case for software systems is not at all the same. Failure events are sometimes quite visible in terms of catastrophic collapses of a system. More often than not, the actual failure event will have occurred a considerable time before its effect is noted. In most cases it is simply not possible to determine with any certainty just when the actual failure occurred on a real-time clock. The most simple example of this improbability of measuring the time between failures of a program may be found in a program that hangs in an infinite loop. Technically the failure event happened on entry to the loop. The program, however, continues to execute until it is killed. This may take seconds, minutes, or hours depending on the patience and/or attentiveness of the operator. As a result, the accuracy of the actual measurement of time intervals is a subject never mentioned in most software validation studies. Because of this, these models are notoriously weak in their ability to predict the future reliability of software systems. A model validated on gaussian noise will probably not do well in practical applications. The bottom line for the measurement of time between failures in software systems is that we cannot measure with any reasonable degree of

accuracy these time intervals. This being the case, we then must look to new metaphors for software systems that will permit us to model the reliability of these systems based on things that we *can* measure with some accuracy.

Yet another problem with the hardware adaptive approach to software reliability modeling is that the failure of a computer software system is simply not time dependent. A system can operate without failure for years and then suddenly become very unreliable based on the changing functions that the system must execute. Many university computer centers experienced this phenomenon in the late 1960s and early 1970s when there was a sudden shift in computer science curricula from programming languages such as FORTRAN that had static run time environments to ALGOL derivatives such as Pascal and Modula that had dynamic run time environments. From an operating system perspective, there was a major shift in the functionality of the operating system exercised by these two different environments. As the shift was made to the ALGOL like languages, latent code in the system, specifically those routines that dealt with memory management, that had not been executed overly much in the past now became central to the new operating environment. This code was both fragile and untested. The operating systems that had been so reliable began to fail like cheap light bulbs.

A new metaphor for software systems would focus on the functionality that the code is executing and not the software as a monolithic system. In computer software systems, it is the functionality that fails. Some functions may be virtually failure free while other functions will collapse with certainty whenever they are executed. The focus of this report is on the notion that it is possible to measure the activities of a system as it executes its various functions and characterize the reliability of the system in terms of these functionalities.

Each program function may be thought of as having an associated reliability estimate. We may chose to think of the reliability of a system in these functional terms. Users of the software system, however, have a very different view of the system. What is important to the user is not that a particular function is fragile or reliable, but rather whether the system will *operate* to perform those actions that the user will want the system to perform correctly. From a user's perspective, it matters not, then, that certain functions are very unreliable. It only matters that the functions associated with the user's actions or operations are reliable. The classical example of this idea was the expressed by the authors of the early UNIX utility programs. In the last paragraph of the documentation for each of these utilities was a list of known bugs for that program. In general, these bugs were not a problem. Most involved aspects of functionality that the typical user would never exploit.

From a functional viewpoint, a program may be viewed as a set of program modules that are executing a set of mutually exclusive functions. If the program executes a functionality consisting of a subset of these modules that are fault-free, it will never fail no matter how long it executes this functionality. If, on the other hand, the program is executing a functionality that contains fault-laden modules, there is a very good

likelihood that it will fail whenever that functionality is expressed. Further, it will fail with certainty when the right aspects of functionality are expressed. Another significant problem in the determination of the reliability of a software system is the precise determination of the failure event. The failure event, and the circumstances that surround the failure, have proven to be most elusive concepts. This investigation will explore an alternative view of software reliability together with a mechanism for the collection of the data necessary to understand the failure.

The main problem in the understanding of software reliability from this new perspective is getting the granularity of the observation right. Software systems are designed to implement each of their functionalities in one or more code modules. In some cases there is a direct correspondence between a particular program module and a particular functionality. That is, if the program is expressing that functionality, it will execute exclusively in the module in question. In most cases, however, there will not be this distinct traceability of functionality to modules. The functionality will be expressed in many different code modules. It is the individual code module that fails. A code module will, of course, be executing a particular functionality when it fails. We must understand that it is the functionality that fails.

As a program is exercising any one of its many functionalities in the normal course of operation of the program, it will apportion its time across this set of functionalities. The proportion of time that a program spends in each of its functionalities is the *functional profile* of the program. Further, within the functionality, it will apportion its activities across one to many program modules. This distribution of processing activity is represented by the concept of the execution profile. In other words, if we have a program structured into *n* distinct modules, the *execution profile* for a given functionality will be the proportion of program activity for each program module while the function was being expressed.

As the discussion herein unfolds, we will see that the key to understanding program failure events is the direct association of these failures to execution events with a given functionality. A Markovian stochastic process will be used to describe the transition of program modules from one to another as a program expresses a functionality. From these observations, it will become fairly obvious just what data will be needed to describe accurately the reliability of the system. In essence, the system will essentially be able to appraise us of its own health. The reliability modeling process is no longer something that will be performed *ex post facto*. It may be accomplished dynamically while the program is executing. The goal of this three-year project has been to develop a methodology that will permit the modeling of the reliability of program functionality. This methodology will then be used to develop notions of design *robustness* in the face of departures from design functional profiles.

The failure of a software system is dependent only on what the software is currently doing: its functionality. If a program is currently executing a functionality that is expressed in terms of a set of fault-free modules, this functionality will certainly execute indefinitely without any likelihood of failure. A program may execute a sequence of fault-prone modules and still not fail. In this case, the faults may lie in

a region of the code that is not likely to be expressed during the execution of a function. A failure event can only occur when the software system executes a module that contains faults. If a functionality is never selected that drives the program into a module that contains faults, then the program will never fail. Alternatively, a program may well execute successfully in a module that contains faults just as long as the faults are not expressed.

Some of the problems that have arisen in past attempts at software reliability determination all relate to the fact that their perspective has been distorted. Programs do not wear out over time. If they are not modified, they will certainly not improve over time, nor will they get less reliable over time. The only thing that really impacts the reliability of a software system is what the system is doing at the moment. A program may work very well for a number of years based on the functions that it is asked to execute. This same program may suddenly become quite unreliable if its functionality is changed by the user.

By keeping track of the state transitions from module to module and function to function we may learn exactly where a system is fragile. This information coupled with the functional profile will tell us just how reliable the system will be when we use it as specified. Programs make transitions from module to module as they execute. These transitions may be observed. Transitions to program modules that are fault-laden will result in an increased probability of failure. We can model these transitions as a stochastic process. Ultimately, by developing a mathematical description for the behavior of the software as it transitions from one module to another driven by the functionalities that it is performing, we can describe the reliability of the functionality. The software system is the sum of its functionalities. If we can know the reliability of the functionalities and how the system apportions its time among these functionalities, we can then know the reliability of the system.

Ongoing investigations into the etiology of software failures in the Space Shuttle PASS has provided substantial insight into the measurement of the reliability of this system. This has led to the conclusion that it is not the software system that fails: it is the software system executing a particular functionality that fails. From this new perspective, the sequential execution of program functions may be modeled as a stochastic process. In particular, the program functionalities are physically expressed within a program as subtrees of modules in a program call-tree hierarchy. The transitions between the program modules in a pairwise fashion may be represented in a transition matrix of a Markov process. Program failures are represented by an absorbing state in the transition matrix. This view of reliability permits the dynamic estimation of the parameters of the underlying multinomial probability distribution representing the transition between program modules. This use of the multinomial probability distribution is particularly convenient in that it has a Dirichlet distribution as its natural conjugate family. Thus, a Bayesian approach may be employed so that each step or epoch in the dynamic operation of a system provides incremental information as to the evolving reliability assessment of the program.

A Formal Description of Program Operation

To assist in the subsequent discussion of program functionality, it will be useful to make this description somewhat more precise by introducing some notation conveniences. Assume that the software system *S* was designed to implement a specific set of mutually exclusive functionalities *F*. Thus, if the system is executing a function $f \in F$ then it cannot be expressing elements of any other functionality in *F*. Each of these functions in *F* was designed to implement a set of software specifications based on a user's requirements. From a user's perspective, this software system will implement a specific set of operations, *O*. This mapping from the set of user-perceived operations, *O*, to a set of specific program functionalities is one of the major functions in the software specification process.

Each operation that a system may perform for a user may be thought of as having been implemented in a set of functional specifications. There may be a one-to-one mapping between the user's notion of an operation and a program function. In most cases, however, there may be several discrete functions that must be executed to expresses the user's concept of an operation. For each operation, *o*, that the system may perform, the range of functionalities, *f*, must be well known. Within each operation, one or more of the system's functionalities will be expressed. For a given operation *o*, these expressed functionalities are those with the property

$$F^{(o)} = \{ f : F \mid \forall \text{ IMPLEMENTS}(o, f) \}$$

It is possible, then, to define a relation IMPLEMENTS over $O \times F$ such that IMPLEMENTS(*o, f*) is true if functionality *f* is used in the specification of an operation, *o*. For each operation $o \in O$, there is a relation p' over $O \times F$ such that $p'(o, f)$ is the proportion of activity assigned to functionality *f* by operation *o*. An example of the IMPLEMENTS relation for two operations implemented in four specified functions is shown in Table 23. In this table, we can see that functions f_1 and f_2 are used to implement the operation o_1.

Table 23. Example of the IMPLEMENTS Relation

$O \times F$	f_1	f_2	f_3	f_4
o_1	T	T		
o_2		T	T	T

In Table 24, there is an example of the relation p'. These numbers represent the proportion of time each of the functions will execute under each of the operations.

Table 24. Example of the p' Relation

$p'(o, f)$	f_1	f_2	f_3	f_4
o_1	0.2	0.8	0	0
o_2	0	0.4	0.4	0.2

The software design process is strictly a matter of assigning functionalities in *F* to specific program modules $m \in M$, the set of program modules. The design process may be thought of as the process of defining a set of relations, ASSIGNS over $F \times M$ such that ASSIGNS(*f, m*) is true if functionality *f* is expressed in module *m*. For a given software system, *S*, let *M* denote the set of all program modules for that system. For each function $f \in F$, there is a relation *p* over $F \times M$ such that $p(f, m)$ is the proportion of execution events of module *m* when the system is executing function *f*. Table 25 shows an example of the ASSIGNS relation for the four functions presented in Table 23. In this example we can see the function f_1 has been implemented in the program modules m_1, m_2 and m_4. One of these modules, m_1, will be invoked regardless of the functionality. It is common to all functions. Other program modules, such as m_2, are distinctly associated with a single function.

Table 25. Example of the ASSIGNS Relation

$F \times M$	m_1	m_2	m_3	m_4	m_5	m_6
f_1	T	T		T		
f_2	T		T		T	
f_3	T		T			T
f_4	T		T		T	T

In Table 26, there is an example of the relation *p***.** These numbers represent the proportion of time each of the functions will execute in each of the program modules. The row marginal values represent the total proportion of time allocated to each of the functions. These are the same values as the column marginals of Table 24. Similarly, the column marginal values of Table 26 represent the proportion of time distributed across each of the six program modules.

Table 26. Example of the p Relation

$p(f,m)$	m_1	m_2	m_3	m_4	m_5	m_6
f_1	1	1	0	1	0	0
f_2	1	0	1	0	0.1	0
f_3	1	0	0.5	0	0	0.3
f_4	1	0	1	0	0.4	0.1

There is a relationship between program functionalities and the software modules that they will cause to be executed. These program modules will be assigned to one of three distinct sets of modules that, in turn, are subsets of *M*. Some modules may execute under all of the functionalities of *S*. This will be the set of common modules. The main program is an example of such a module that is common to all operations of the software system. Essentially, program modules will be members of one of two mutually exclusive sets. There is the set of program modules M_c of common modules and the set of modules M_F that are invoked only in response to the execution of a particular function. The set of common modules, $M_c \subset M$ is defined as those modules that have the property

$$M_c = \{m: M \mid \forall f \in F \bullet \text{ASSIGNS(f,m)}\}$$

All of these modules will execute regardless of the specific functionality being executed by the software system.

Yet another set of software modules may or may not execute when the system is running a particular function. These modules are said to be potentially involved modules. The set of potentially involved modules is.

$$M_p^{(f)} = \{m : M_F \mid \exists f \in F \bullet \text{ASSIGNS}(f,m) \wedge 0 < p(f,m) < 1\}$$

In other program modules, there is extremely tight binding between a particular functionality and a set of program modules. That is, every time a particular function, *f*, is executed, a distinct set of software modules will always be invoked. These modules are said to be indispensably involved with the functionality *f*. This set of indispensably involved modules for a particular functionality, *f*, is the set of those modules that have the property that

$$M_i^{(f)} = \{m : M_F \mid \forall f \in F \bullet \text{ASSIGNS}(f,m) \Rightarrow p(f,m) = 1\}$$

As a direct result of the design of the program, there will be a well-defined set of program modules, M_f, that might be used to express all aspects of a given functionality, *f*. These are the modules that have the property that

$$m \in M_f = M_c \cup M_p^{(f)} \cup M_i^{(f)}$$

From the standpoint of software design, the real problems in understanding the dynamic behavior of a system are not necessarily attributable to the set of modules, M_i, that are tightly bound to a functionality or to the set of common modules, M_c, that will be invoked for all executing processes. The real problem is the set of potentially invoked modules, M_p. The greater the cardinality of this set of modules, the less certain we may be about the behavior of a system performing that function. For any one instance of execution of this functionality, a varying number of the modules in M_p may execute.

For each system *S* there is a call graph that shows the transition of program control from one program module to another. To simplify this discussion, let us define a relation CALLS over $M_f \times M_f$. A sample call graph for the hypothetical program described in Tables 23-26 is shown in Figure 9.

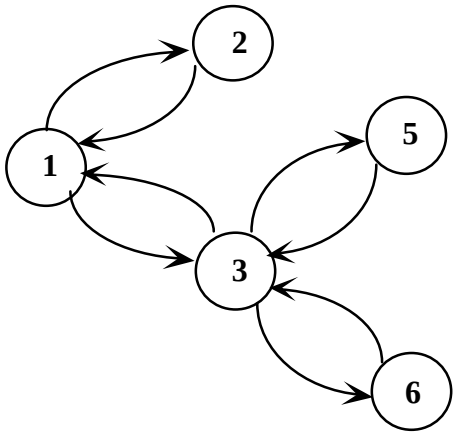


Figure 9. Sample call graph

This call graph represents the transfer of program control to a sequence of program modules, starting with the initial node, 1.

A Stochastic Description of Program Operation

When a program begins the execution of a functionality, say f_4 from Tables 25 and 26, we may envision this beginning as the start of a stochastic process. From the sample call graph shown in Figure 9, we may then construct a probability adjacency matrix, *P*, whose entries represent the transition probability from

each module to another module at each epoch in the execution process while the function, f_4, is executing. Thus, the element $p_{ij}^{(n)}$ of this matrix on the n^{th} epoch are the probabilities that $\mathrm{CALLS}(m_i, m_j)$ is true for that epoch. For example, consider the following matrix that contains the probabilities for the state transitions for the four modules, m_1, m_3, m_5, and m_6, that constitute the set M_{f_4}

$$\boldsymbol{P} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0.2 & 0 & 0.3 & 0.5 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

The first row of this matrix represents the transition probability of m_1 to m_1, m_3, m_5, and m_6, the second row the transition probabilities for m_3, and so forth.

Execution of the Call Graph as a Markov Process

The transition from one module to another may be described as a stochastic process. In which case we may define an indexed collection of random variables $\{X_t\}$, where the index *t* runs through a set of non-negative integers, $t = 0, 1, 2, \cdots$ representing the epochs of the process. At any particular epoch the software is found to be executing exactly one of its *M* modules. The fact of the execution occurring in a particular module is a *state* of the system. For this software system, the system is found in exactly one of a finite number of mutually exclusive and exhaustive states that may be labeled $0, 1, 2, \cdots, M$. In this representation of the system, there is a stochastic process $\{X_t\}$, where the random variables are observed at epochs $t = 0, 1, 2, \cdots$ and where each random variable may take on any one of the $(M+1)$ integers, from the state space $A = \{0, 1, 2, \cdots, M\}$.

A stochastic process $\{X_t\}$ is a Markov chain if it has the property that

$$\Pr[X_{t+1} = j | X_t = i_t, X_{t-1} = i_{t-1}, X_{t-2} = i_{t-2}, \cdots, X_0 = i_0] = \Pr[X_{t+1} = j | X_t = i_t]$$

for any epoch $t = 0, 1, 2, \cdots$. and all states $i_0, i_1, \cdots, i_t$ in the state space *A*. This is equivalent to saying that the conditional probability of executing any module at any future epoch is dependent only on the current state of the system. The conditional probabilities $\Pr[X_{t+1} = j | X_t = i_t]$ are called the transition probabilities. In that this nomenclature is somewhat cumbersome, let $p_{ij}^{(n)} = \Pr[X_n = j | X_{n-1} = i]$. Within the execution of a given functionality, the behavior of the system is static. That is, the transition probabilities do not change from one epoch to another. Thus, $\Pr[X_{t+1} = j | X_t = i_t] = \Pr[X_1 = j | X_0 = i_0]$ for *i, j,* in *S*, which is an additional condition of a Markov process.

Since the $p_{ij}^{(n)}$ are conditional probabilities, it is clear that

$$p_{ij}^{(n)} \geq 0, \text{ for all } i,j \text{ in } A, \; n = 0, 1, 2, \cdots$$

and

$$\sum_{j=0}^{M} p_{ij}^{(n)} = 1, \text{ for all } i \text{ in } A \text{ and } n = 0,1,2,\cdots$$

If we use the nomenclature $\mathbf{P}$ to denote the matrix of one-step transition probabilities at the initial epoch, then the system at the $n+1^{st}$ epoch can be obtained from the expression

$$\boldsymbol{P}^{(n)} = \boldsymbol{P}^{n} = \boldsymbol{P} \cdot \boldsymbol{P}^{n-1}$$

What we would like to ascertain is the unconditional probability of being in a particular module at a particular epoch. To find this conditional probability let us first observe that

$$\Pr[X_{t+1} = j | X_t = i_t] = \Pr[X_1 = j | X_0 = i_0]$$

It is clear that the unconditional probability of executing a module *j* and epoch *n*, then, is dependent only on the initial state of the system. Thus,

$$\Pr[X_n = j] = \sum_{i=0}^{M} p_{ij}^{(n)} \Pr[X_0 = i]$$

Interestingly enough, for all software systems there is a distinguished module, the main program module, that will always receive execution control from the operating system. If we denote this main program as module 0, then

$$\Pr[X_0 = 0] = 1 \text{ and } \Pr[X_0 = i] = 0 \text{ for } i = 1,2,\cdots,M$$

We can see, then, that the unconditional probability of executing in a particular module *j* is

$$\Pr[X_n = j] = p_{ij}^{(n)} \Pr[X_0 = 0] = p_{ij}^{(n)}$$

State Transitions for Fault-Free Modules

The granularity of the term epoch is now of interest. An epoch begins with the onset of execution in a particular module and ends when control is passed to another module. The measurable event for modeling purposes is this transition among the program modules. We will count the number of calls from a module and the number of returns to that module. Each of these transitions to a different program module from the one currently executing will represent an incremental change in the epoch number. Computer programs executing in their normal mode will make state transitions between program modules rather rapidly. In terms of real clock time, many epochs may elapse in a relatively short period. Thus, we will now turn our attention to the long-term behavior of the software system.

There are three types of Markov processes that may be used to describe or model the stochastic behavior of a software system depending on the nature of the interaction of the program modules. It is a characteristic of the first two types of Markov processes that $p_{ij} \neq 0 \Leftrightarrow p_{ji} \neq 0$ for $i, j = 1, 2, \cdots, M$ and $i \neq j$.

The first and least complex case is the description of a set of program modules that are not recursive. If there is no recursion in the system, then $p_{ii}^{(0)} = 0$ for $i = 1, 2, \cdots, M$. This will result in a periodic Markov process. The periodicity, *d*, of this process is equal to the number of eigenvalues of *P* of modulus 1.

The second distinct Markov process is one that does have recursive functions in it. If there are one or more recursive modules in the system, then $0 < p_{ii}^{(0)} < 1$ for at least one $i = 1, 2, \cdots, M$. The resulting Markov process is ergodic and the steady-state behavior of the process may be characterized by a single transition matrix. It can be shown that

$$\lim_{n \to \infty} p_{ij}^{(n)} = \tau_j$$

where the τ_j's satisfy the following steady state equations:

$$\begin{aligned} &\tau_j > 0, \\ &\tau_j = \sum_{i=0}^{M} \tau_i p_{ij}^{(0)}, \text{ for } j \text{ in } A, \\ &\sum_{j=0}^{M} \tau_j = 1. \end{aligned}$$

The τ_j's are the steady-state probabilities of the Markov chain. They represent the long-term distribution of system activity in each of the program modules.

State Transitions for Failure-Prone Modules

The third type of Markov process is one that has an absorbing state, in which case $p_{ii}^{(0)} = 1$ for at least one $i = 1, 2, \cdots, M$. An example of such a system is a call tree that has a module that always exits to the operating system. Once this state has been entered, no other state is reachable from this absorbing state. We will use this notion of an absorbing state to model the failure of a system. In this case, we will consider the failure of a program module to be the transition from that module to the absorbing failure state.

While the first two Markovian processes discussed above dealt with a program that would not fail, we now wish to examine the potential for modeling the failure of a software system. When a program module fails, we can imagine that the module has made a transition to an absorbing state, a failure state, in the Markov transition matrix. Thus, every program may be thought to have a virtual module representing the failed state of program. When this virtual module receives control, it will not relinquish it. The transition matrix

for this new model is augmented by an additional row and a new column. For a program with *M* modules, let the error state be represented by a new state, $T = M + 1$. For this new state,

$$p_{Tj}^{(n)} \begin{cases} = 0 & \text{for all } j = 1, 2, \ldots, M \\ = 1 & \text{for } j = T \end{cases}, \; n = 0, 1, 2, \cdots$$

This represents the augmented row of the new transition matrix. Each row in the transition matrix will be augmented by a new column entry $p_{iT}^{(n)}$ for $i = 1, 2, \cdots, M$, where $p_{iT}^{(n)}$ represents the probability of the failure of the i^{th} module in the n^{th} epoch. When a program dies, it is the result of a fault in one or more of its modules. Not all modules are equally likely to lead to the failure event. The fault proneness of the module is distinctly related to measurable software attributes. When program modules are executed that are fault-prone, they are much more likely to fail than those that are not fault-prone. We seek a forecasting or prediction mechanism that will capitalize on this understanding.

The Profiles of Software Dynamics

When a program is executing a functionality it will apportion its activities among a set of modules. As such it will transition from one module to the next on a call (or return) sequence. Each module called in this call sequence will have an associated call frequency. When the software is subjected to a series of unique and distinct functional expressions, there will be a different Markov chain for each of the user's operations in that each will implement a different set of functions that will, in turn, invoke possibly different sets of program modules. For each of the functions it is clear that the transition matrix $\boldsymbol{P}$ may be partitioned into a submatrix $\boldsymbol{P}^f$ where the rows and columns of this matrix represent the elements of M_f.

As a particular function is being executed, it will be necessary to record the transition from one module to another. These data may be recorded in a matrix, $\boldsymbol{A}$, whose elements represent the frequency of transitions from one module to another. After a sequence of *n* epochs, the element, a_{ij}, of the matrix, $\boldsymbol{A}$, will contain the number of calls into module *j* from module *i* during the *n* epochs. This matrix will have non-zero entries for all of the modules in $M_c \cup M_i^{(f)}$. It may or may not have non-zero entries for the elements of $M_p^{(f)}$. As an example of this notion, let us suppose that function f_4 was run on two separate occasions. On the first occasion, the function, f_4, executed for a total of 420 epochs. The resulting calls are represented by the matrix $\boldsymbol{A}_1$. All four modules, m_1, m_3, m_5, and m_6, that constitute the set M_{f_4} were executed in this first example.

$$\boldsymbol{A}_1 = \begin{pmatrix} 0 & 40 & 0 & 0 \\ 40 & 0 & 120 & 50 \\ 0 & 120 & 0 & 0 \\ 0 & 50 & 0 & 0 \end{pmatrix} \qquad \boldsymbol{A}_2 = \begin{pmatrix} 0 & 18 & 0 & 0 \\ 18 & 0 & 0 & 75 \\ 0 & 0 & 0 & 0 \\ 0 & 75 & 0 & 0 \end{pmatrix}$$

The second matrix, $\boldsymbol{A}_2$, represents the execution of the same functionality, f_4, for 186 epochs, but this time the module $m_5 \in M_p^{(f_4)}$ did not execute. The two matrices, $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ are symmetrical. The upper triangular portion of these matrices represents the call from each module and the lower triangle represents the returns from each call. The activity of the system in and out of modules will *profile* the activity of the functions that software was designed to perform.

Functional Profiles

When a software system is constructed by the software developer, it is designed to fulfill a set of specific functional requirements. The user will run the software to perform a set of perceived operations. In this process, the user will typically not use all of the functionalities with the same probability. The functional profile of the software system is the set of unconditional probabilities of each of the functionalities *F* being executed by the user. Let *Y* be a random variable defined on the indices of the set of elements of *F*. Then, $o_k = \Pr[Y = k], k = 1, 2, \ldots, \#\{F\}$ is the probability that the user is executing program functionality *k* as specified in the functional requirements of the program and $\#\{F\}$ is the cardinality of the set of functions. A program executing on a serial machine can only be executing one functionality at a time. The distribution of *o*, then, is multinomial for programs designed to fulfill more than two specific functions. The prior knowledge of this distribution of functions should guide the software design process.

Execution Profiles

When a program is executing a given functionality, say f_k, it will distribute its activity across the set of modules, M_{f_k}. At any arbitrary epoch, *n*, the program will be executing a module $m_i \in M_{f_k}$ with a probability, $u_{ik} = \Pr[X_n = i | Y = k]$. The set of conditional probabilities $u_{\bullet k}$ where $k = 1, 2, \ldots, \#\{F\}$ constitute the execution profile for function f_k. As was the case with the functional profile, the distribution of the execution profile is also multinomial for a software system consisting of more than two modules. As a matter of the design of a program, there may be a non-empty set $M_p^{(f)}$ of modules that may or may not be executed when a particular functionality is exercised. This will, of course, cause the cardinality of the set M_f to vary. A particular execution may not invoke any of the modules of $M_p^{(f)}$. On the other hand, all of the modules may participate in the execution of that functionality. This variation in the cardinality of M_f within the execution of a single functionality will contribute significantly to the amount of test effort that will be necessary to test such a functionality.

Each operation will implement a subset of functionalities, i.e. $F_e^{(o)} \subset F$. As each operation is run to completion, it will generate an execution profile. This execution profile may represent the results of the execution of one or more functions. Most operations, though, do not exercise precisely one functionality. Rather, they may apportion time across a number of functionalities. For a given operation, let *l* be a proportionality constant. Then, $0 \le l_k \le 1$ will represent the proportion of epochs that will be spent

executing the k^{th} functionality in $F_e^{(o)}$. Thus an operational profile of a set of modules will represent a linear combination of the conditional probabilities, u_{ik} as follows:

$$p_i = \sum_{f_k \in F_e^{(o)}} l_k u_{ik}.$$

Module Profiles

The manner in which a program will exercise its many modules as the user chooses to execute the functionalities of the program is determined directly by the design of the program. Indeed, this mapping of functionality onto program modules is the overall objective of the design process. The *module profile*, *q*, is the unconditional probability that a particular module will be executed based on the design of the program. It is derived through the application of Bayes' rule. First, the joint probability that a given module is executing and the program is exercising a particular function is given by

$$\Pr[X_n = j \cap Y = k] = \Pr[Y = k] \Pr[X_n = j | Y = k] = o_k u_{ik}$$

where *j* and *k* are defined as before. Thus, the unconditional probability, q_i of executing module *j* under a particular design is

$$\begin{aligned} q_i &= \Pr[X_n = i] \\ &= \sum_k \Pr[X_n = i \cap Y = k) \\ &= \sum_k o_k u_{ik} \end{aligned}$$

As was the case for the functional profile and the execution profile, only one module can be executing at any one time. Hence, the distribution of *q* is also multinomial for more than two modules.

The Transition Probabilities for Functions

The final profile consideration will be the determination of the transition probabilities $p_{ij}^{(0)} = \Pr[X_1 = j \mid X_0 = i]$ of $\boldsymbol{P}^0$. Each row *i* of *P* represents the probability of the transition to a new state *j* given that the program is currently in state *i*. These are mutually exclusive events. The program may only transfer control to exactly one other program module. Under this assumption, the conditional probabilities that are the rows of $\boldsymbol{P}^0$, also have the property that they are distributed multinomially. They profile the transitions from one state to another.

Estimates for Transition Probabilities and Profiles

The focus will now shift to the problem of understanding the nature of the distribution of the probabilities for various profiles. We have so far come to recognize these profiles in terms of their multinomial nature. The multinomial distribution is useful for representing the outcome of an experiment involving a set of mutually exclusive events. Let $S = \bigcup_{i=1}^{M} S_i$ where S_i is one of M *mutually* exclusive sets of events. Each of these events would correspond to a program executing a particular module in the total set of program modules. Further, let $\Pr(S_i) = w_i$ and

$$w_T = 1 - w_1 - w_2 - \cdots - w_M,$$

under the condition that $T = M + 1$, as defined earlier, in which case w_i is the probability that the outcome of a random experiment is an element of the set S_i. If this experiment is conducted over a period of n trials then the random variable X_i will represent the frequency of S_i outcomes. In this case, the value, n, represents the number of transitions from one program module to the next. Note that

$$X_T = n - X_1 - X_2 - \cdots - X_M$$

This particular distribution will be useful in the modeling of a program with a set of k modules. During a set of n program steps, each of the modules may be executed. These, of course, are mutually exclusive events. If module i is executing, then module j cannot be executing.

The multinomial distribution function with parameters n and $\boldsymbol{w} = (w_1, w_2, \ldots, w_T)$ is given by

$$f(\boldsymbol{x} \mid n, \boldsymbol{w}) = \begin{cases} \dfrac{n!}{\prod_{i=1}^{k-1} x_i!} w_1^{x_1} w_2^{x_2} \cdots w_M^{x_M}, & (x_1, x_2, \cdots, x_M) \in S \\ 0 \text{ elsewhere} \end{cases}$$

where x_i represents the frequency of execution of the i^{th} program module.

The expected values for the x_i are given by

$$E(x_i) = \bar{x}_i = nw_i,\ i = 1, 2, \ldots, k,$$

the variances by

$$Var(x_i) = nw_i(1 - w_i)$$

and the covariance by

$$Cov(w_i, w_j) = -nw_i w_j \, , i \neq j$$

We would like to come to understand, for example, the multinomial distribution of a program's execution profile while it is executing a particular functionality. The problem here is that every time a program is run we will observe that there is some variation in the profile from one execution sample to the next. It will be difficult to estimate the parameters $\boldsymbol{w} = (w_1, w_2, \ldots, w_T)$ for the multinomial distribution of the execution profile. Rather than estimating these parameters statically, it would be far more useful to us to get estimates of these parameters dynamically as the program is actually in operation, hence the utility of the Bayesian approach.

To aid in the process of characterizing the nature of the true underlying multinomial distribution, let us observe that the family of Dirichlet distributions is a conjugate family for observations that have a multinomial distribution. The p.d.f. for a Dirichlet distribution, $D(\boldsymbol{\alpha}, \alpha_T)$, with a parametric vector $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_M)$ where $(\alpha_i > 0;\ i = 1, 2, \ldots, M)$ is

$$f(w \mid \boldsymbol{\alpha}) = \frac{\Gamma(\alpha_1 + \alpha_2 + \cdots + \alpha_M)}{\prod_{i=1}^{M} \Gamma(\alpha_i)} w_1^{\alpha_1 - 1} w_2^{\alpha_2 - 1} \cdots w_M^{\alpha_M - 1}$$

where $(w_i > 0;\ i = 1, 2, \ldots, M)$ and $\sum_{i=1}^{M} w_i = 1$. The expected values of the w_i are given by

$$E(w_i) = \mu_i = \frac{\alpha_i}{\alpha_0} \tag{1}$$

where $\alpha_0 = \sum_{i=1}^{T} \alpha_i$. In this context, α_0 represents the total epochs. The variance of the w_i is given by

$$Var(w_i) = \frac{\alpha_i(\alpha_0 - \alpha_i)}{\alpha_0^2(\alpha_0 + 1)}, \tag{2}$$

and the covariance by

$$Cov(w_i, w_j) = \frac{\alpha_i \alpha_j}{\alpha_0^2(\alpha_0 + 1)}$$

Within the set of expected values $\mu_i, i = 1, 2, \ldots, T$, not all of the values are of equal interest. We are interested, in particular, in the value of μ_T. This will represent the probability of a transition to the terminal failure state from a particular program module. So that we might use this value for

our succeeding reliability prediction activities, it will be useful to know how good this estimate is. To this end, we would like to set 100(1-α)% confidence limits on the estimate. For the Dirichlet distribution, this is not clean. To simplify the process of setting these confidence limits, let us observe that if $\boldsymbol{w} = (w_1, w_2, \ldots, w_M)$ is a random vector having the *M*-variate Dirichlet distribution, $D(\boldsymbol{\alpha}, \alpha_T)$, then the sum $z = w_1 + \cdots + w_M$ has the beta distribution,

$$f_\beta(z \mid \gamma, \alpha_T) = \frac{\Gamma(\gamma + \alpha_T)}{\Gamma(\gamma)\Gamma(\alpha_T)} z^\gamma (1 - z)^{\alpha_T}$$

or alternately

$$f_\beta(w_T \mid \gamma, \alpha_T) = \frac{\Gamma(\gamma + \alpha_T)}{\Gamma(\gamma)\Gamma(\alpha_T)} (1 - w_T)^\gamma (w_T)^{\alpha_T},$$

where $\gamma = \alpha_1 + \alpha_2 + \cdots + \alpha_M$.

Thus, we may obtain 100(1-α)% confidence limits for

$$\mu_T - a \le \mu_T \le \mu_T + b$$

from

$$F_\beta(\mu_T - a \mid \gamma, \alpha_T) = \int_0^{\mu_T - a} f_\beta(w_T \mid \gamma, \alpha_T) dw = \frac{\alpha}{2} \tag{3}$$

and

$$F_\beta(\mu_T + b \mid \gamma, \alpha_T) = \int_0^{\mu_T + b} f_\beta(w_T \mid \gamma, \alpha_T) dw = 1 - \frac{\alpha}{2} \tag{4}$$

Where this computation is inconvenient, let us observe that the cumulative beta function, F_β, can also be obtained from existing tables of the cumulative binomial distribution, F_b by making use of the knowledge that

$$F_b(\gamma \mid \mu_T - a, \gamma + \alpha_T) = F_\beta(\mu_T - a \mid \gamma, \alpha_T)$$

and

$$F_b(\alpha_T \mid 1 - (\mu_T + b), \gamma + \alpha_T) = F_\beta(\mu_T + b \mid \gamma, \alpha_T)$$

The value of the use of the Dirichlet conjugate family for modeling purposes is twofold. First, it permits us to estimate the probabilities of the module transitions directly from the observed transitions. Secondly, we are able to obtain revised estimates for these probabilities as the

observation process progresses. Let us now suppose that we wish to model the behavior of a software system whose execution profile has a multinomial distribution with parameters n and $\boldsymbol{W} = (w_1, w_2, \ldots, w_M)$ where n is the total number of observed module transitions and the values of the w_i are unknown. Let us assume that the prior distribution of $\boldsymbol{W}$ is a Dirichlet distribution with a parametric vector $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_M)$ where $(\alpha_i > 0;\ i = 1, 2, \ldots, M)$. Then the posterior distribution of $\boldsymbol{W}$ for the behavioral observation $\boldsymbol{X} = (x_1, x_2, \ldots, x_M)$ is a Dirichlet distribution with parametric vector $\boldsymbol{\alpha}^* = (\alpha_1 + x_1, \alpha_2 + x_2, \ldots, \alpha_M + x_M)$. As an example, suppose that we now wish to model the behavior of a large software system with such a parametric vector. As the system makes sequential transitions from one module to another, the posterior distribution of $\boldsymbol{W}$ at each transition will be a Dirichlet distribution. Further, for $i = 1, 2, \ldots, T$ the i^{th} component of the augmented parametric vector $\boldsymbol{\alpha}$ will be increased by 1 unit each time module m_i is executed.

An Example

Consider our simple example of the sample program executing the function f_4. Initially we will assume that we have little or no information as to the probabilities for $\boldsymbol{P}^0$. In the face of incomplete information we may choose to assume that all state transitions are equiprobable. Hence, the prior probability distributions for $\boldsymbol{P}^0$ will look like this:

$$\boldsymbol{P}^0 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0.33 & 0 & 0.33 & 0.33 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

This would correspond to an initial observation matrix $\boldsymbol{A}_0$ where each outcome has hypothetically been observed once. Thus,

$$\boldsymbol{A}_0 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Now let us assume that we have actually executed this function and have observed the transfer of control among the modules as indicated in the matrix $\boldsymbol{A}_1$ from above. The row marginals for this matrix enumerate the total control transfers from each module. If we let $a_{i0} = \sum_{j=1}^{M} a_{ij}$ then from (1) above, the expected values of the posterior distribution for the rows of $\boldsymbol{P}^0$ may be obtained

from

$$E(p_{ij}^{0}) = \frac{a_{ij}}{a_{i0}}$$

giving

$$\boldsymbol{P}^{0'} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0.19 & 0 & 0.57 & 0.24 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

The variances of the transition probabilities for the second row are only those that are interesting in this example. They may be obtained from (2) and are (0.00073, 0, 0.00116. 0.00086). The eigenvalues of this matrix are (0, 1, -1, 0) indicating that this structure has a periodicity of 2.

If we now continue running the program and observe the new module transitions shown in the matrix $\boldsymbol{A}_2$. The total transitions observed so far will be

$$\boldsymbol{A}_3 = \begin{pmatrix} 0 & 59 & 0 & 0 \\ 59 & 0 & 121 & 126 \\ 0 & 121 & 0 & 0 \\ 0 & 126 & 0 & 0 \end{pmatrix}$$

and the new posterior distribution for the rows of $\boldsymbol{P}^0$ would be

$$\boldsymbol{P}^{0''} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0.19 & 0 & 0.40 & 0.41 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

The variances of the second row of this new transition matrix are (0.00051, 0, 0.00079. 0.00080). The variances, then, are diminishing as our information about the behavior of this system improves.

Reliability Estimation

The modeling of reliability of systems will be implemented through the use of an absorbing Markov chain. In this application, we will postulate the existence of a virtual program module

representing the failure of the system. Should control ever be transferred to this module, it will never be returned. Each program module has a non-zero probability of transferring control to this virtual failure module. This probability is directly related to the fault-proneness of the module. We may, in fact, use the functional relationship between software complexity and software faults to derive our prior probabilities for the transition between each program module and the virtual failed state module.

What is important to understand is that each program module is distinctly related to one or more functions. If a function is expressed by a set of modules that are failure-prone, then the function will appear to be failure-prone. If, on the other hand, a function is expressed by a set of modules in a call tree that are fault-free, this function will never fail. The key point is that it is functionality that fails. Not all functions will be executed by a user with the same likelihood. If a user executes unreliable functions consistently, then he will perceive the system to be unreliable. Conversely, if another user were to use the same system but exercise functionalities that were not so likely to fail, then his perceptions of the same system would be very different.

To model the reliability of a software function, we will augment the basic Markov chain to include an absorbing state that represents a virtual program module called the failure state.

Each program module may have a non-zero transition probability to this virtual module. If a module is fault-free, then its transition probability will be zero.

To continue the example from the previous section, let us now augment the transition matrix $\boldsymbol{P}^0$ with a virtual failure state and an equiprobable transition of each module to this state. The new matrix would look like this,

$$\boldsymbol{P}^0 = \begin{pmatrix} 0 & 0.50 & 0 & 0 & 0.50 \\ 0.25 & 0 & 0.25 & 0.25 & 0.25 \\ 0 & 0.50 & 0 & 0 & 0.50 \\ 0 & 0.50 & 0 & 0 & 0.50 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

This would correspond to an initial presumed observation matrix $\boldsymbol{A}_0$ of

$$\boldsymbol{A}_1' = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 4 \end{pmatrix},$$

which is to say that we have little or no information about the initial transition probabilities: each of the outcomes is equally likely. After an arbitrary number of epochs, say 419, the new transition matrix for the process at that point would be

$$\boldsymbol{P}^{0^{419}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

As we observe the system at this point, there is complete certainty of a failure event in each of the program modules, given our initial estimates of failure probabilities.

Now let us consider that we have run the system for the complete 420 epochs of our earlier example. The system has failed three times in this example, once in module 2 and twice in module 3. This will give us a new matrix of observations as follows:

$$\boldsymbol{A}_1^{'} = \begin{pmatrix} 0 & 41 & 0 & 0 & 1 \\ 41 & 0 & 121 & 51 & 2 \\ 0 & 121 & 0 & 0 & 3 \\ 0 & 51 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 7 \end{pmatrix}$$

The new posterior distribution using these new observations would be

$$\boldsymbol{P}^{0^{'}} = \begin{pmatrix} 0 & 0.9762 & 0 & 0 & 0.0238 \\ 0.1907 & 0 & 0.5628 & 0.2372 & 0.0093 \\ 0 & 0.9758 & 0 & 0 & 0.0242 \\ 0 & 0.9808 & 0 & 0 & 0.0192 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The elements of this transition matrix are derived from

$$E(p_{ij}) = \frac{a_{ij}}{a_{0j}}$$

where $a_{0j} = \sum_{i=1}^{T} a_{ij}$.

We are learning that the system is far less failure-prone than our initial uninformed estimates, $\boldsymbol{P}^0$. Also, we are coming to understand that not all modules demonstrate the same failure potential. The probability of a failure from Module 3, for example, is materially less than that of Module 5.

Now let us assume that the system has been driven over an additional interval of 187 epochs. We now observe that there is a non-zero probability of a projected execution of each of these modules based on our revised estimates for the transition probabilities in $\boldsymbol{P}^0$. After the next hypothetical sequence of 187 epochs, our current computed transition matrix for the process at this point would be

$$\boldsymbol{P}^{187\prime} = \begin{pmatrix} 0 & 0.0473 & 0 & 0 & 0.9525 \\ 0.0092 & 0 & 0.0273 & 0.0115 & 0.9518 \\ 0 & 0.0473 & 0 & 0 & 0.9525 \\ 0 & 0.0475 & 0 & 0 & 0.9523 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The observations made during these 187 epochs might show the following events, including one failure attributable to module 2. This matrix of cumulative observations might hypothetically be

$$\boldsymbol{A}_2{}' = \begin{pmatrix} 0 & 59 & 0 & 0 & 1 \\ 59 & 0 & 121 & 126 & 3 \\ 0 & 121 & 0 & 0 & 3 \\ 0 & 126 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 8 \end{pmatrix}$$

It is derived by summing the elements of $\boldsymbol{A}_1$, $\boldsymbol{A}_2$, and $\boldsymbol{A}_3$. This, in turn, would yield a new posterior transition matrix as follows:

$$\boldsymbol{P}^{0\prime\prime} = \begin{pmatrix} 0 & 0.9833 & 0 & 0 & 0.0167 \\ 0.1909 & 0 & 0.3916 & 0.4078 & 0.0097 \\ 0 & 0.9758 & 0 & 0 & 0.0242 \\ 0 & 0.9921 & 0 & 0 & 0.0079 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The value of the failure probability estimates that we will have established in the above manner lies in their usefulness for future predictive ability. This, in turn, is directly related to the confidence intervals that we must establish for these estimates. These confidence intervals may be obtained from equations (3) and (4). For the purposes of this example, the upper and lower 5%

confidence intervals ($\frac{\alpha}{2}$, $1-\frac{\alpha}{2}=0.975$) for the transition probabilities of each module to the absorbing failure state have been compiled in Table 5. The lower confidence intervals are of little interest. In fact, as the number of observed transitions becomes large, these values will approach zero. What is of great interest to us are the upper confidence interval values for μ_T. Functionally these are the values that will have the greatest impact on the evaluation of our modeling. We can see, for example, that the expected value of the failure probability of Module 3 is $\mu_T = 0.0097$. The upper confidence limit for this value is greater by more than a factor of two.

Table 27. Module Reliability Estimates With 5% Confidence Intervals

Interval	Module	$\frac{\alpha}{2}$	μ_T	$1-\frac{\alpha}{2}$
Initial	Module 1	0.0126	0.5000	0.8489
	Module 3	0.0063	0.2500	0.6024
	Module 5	0.0126	0.5000	0.8489
	Module 6	0.0126	0.5000	0.8489
Second	Module 1	0.0006	0.0244	0.0860
	Module 3	0.0011	0.0093	0.0255
	Module 5	0.0049	0.0242	0.0562
	Module 6	0.0005	0.0192	0.0684
Final	Module 1	0.0035	0.0167	0.0880
	Module 3	0.0040	0.0097	0.0278
	Module 5	0.0128	0.0242	0.0781
	Module 6	0.0019	0.0079	0.0424

To this point we have created a mechanism for modeling the transition of each program module to the failure state. In this sense the reliability of each module m_i may be directly determined by the elements, p_{iT}^0, of $\boldsymbol{P}^0$. With the Bayesian approach, we have also established a mechanism for refining our estimates of these reliabilities and establishing a measure of confidence in each of these estimates. This information will now be used to establish the reliability of functions that employ each of the modules in varying degrees. The successive powers of $\boldsymbol{P}^0$ will show the

failure likelihood for each of the modules. This will permit us to postulate on the probability of a failure at some future epoch *n* based on the current estimates of failure probability.

Functional Reliability

Not all states that a system of *M* program modules can get into hold equal fascination for us. In the augmented program model above, we postulated the existence of a virtual program module m_K, a program module representing an error state. This module may or may not be invoked depending on the particular functionality being executed. If a function executes a set of unreliable modules with a high probability, then the function will not be reliable. If, on the other hand, the function executes only highly reliable modules, then the function will be perceived to be reliable. It is a characteristic of each function that it exhibits an execution profile $u_{\bullet k}$. Each module has an associated reliability. Let us define the reliability of the j^{th} module to be $r_j = 1 - p_{jT}^{(0)}$. The expected value for the reliability of the function, then is $\mu_f = E(r_f) = \sum_{j=1`}^{m} u_{\bullet j} r_j$ where $u_{\bullet j}$ is the execution profile for the function as defined earlier and *m* is the number of modules. This reliability estimate, however, is only for a particular functionality. It is derived from the execution profile of a given functionality. Thus, each function has its own independent reliability assessment. That was the original intent of this investigation, to demonstrate a mechanism for the determination of the reliability of program functionality. It is program functions that fail. Some functions are more reliable than others. We can measure reliability at the functional level.

The reliability of the individual functions is dependent on the distribution of $u_{\bullet j}$ for the function. As was indicated earlier, the underlying distribution for $u_{\bullet j}$ is multinomial. We may derive estimates for these probabilities in precisely the same manner that we used for developing the estimates for the conditional probabilities of the transition matrix. We simply need to count the frequency with which each module is executed when a particular function is being executed. Computation for the estimates for the $u_{\bullet j}$ will proceed as above, as will the determination for the confidence intervals for these estimates.

Now we arrive at the real problem in the estimation of the reliability of a functionality. Our long-term ability to understand and/or estimate the execution profile of a system is clouded by the set of modules $M_p^{(f)}$ that may or may not execute when a particular functionality is expressed. It would be arrogant or ignorant to assume that the execution profiles for all functions were stable and knowable. (The worse the design, the more certain this is true). The Bayesian approach to reliability determination is used in that we may use the information currently at our disposal to provide the best estimate as to the future behavior of the system. If the set $M_p^{(f)}$ is empty, then the behavior of the system is quite tractable. In fact, if we were to normalize the elements of the

$\boldsymbol{A}_i'$ cumulative observation matrices by dividing each element by the number of observed epochs, we would find that resulting transition probabilities were quite stable over repeated observations.

The very best way to create a system whose reliability may never be understood is to design the system with a large set of modules $M_p^{(f)}$. The obverse of this coin is a tractable system with little or no variability in its function execution profiles. The reliability of such a system may be assessed quite accurately. This is not to imply that a reliable system is one whose reliability may be measured accurately; quite the contrary. We may conceive a very unreliable system whose behavior may be well understood.

A reliable system is one that *by design* spends a high proportion of its execution time in modules that are not likely to fail. In other words, there is a strong positive correlation between the measures of complexity of a module and its design functional profile. Such fault-prone modules may, in fact, be identified by their intrinsic attributes during the design stage. If a system is carefully designed with these criteria controlling the design process, the resulting software will be reliable. It may, however, not be robust.

A design is robust if it does not suffer a diminution in its functional reliability in the face of departures from its design functional profile. Not all reliable systems are robust. The overall robustness of a system is dependent on how the specific functions implement users' operations. A robust system is one that remains reliable in the face of departures from the design operational profile of the system. Systems may be designed for both reliable and robust operation.

Data Collection for Reliability Estimation: A User's Guide for *DRAT*

It seems pointless to engage in the academic exercise of software reliability modeling without making at least the slightest attempt to discuss the process of measuring the independent variable(s) in these models. The basic premise of this report is that we really cannot measure temporal aspects of program failure. There are, however, certain aspects of program behavior that we can measure and also measure with accuracy. We can measure transitions into and out of program modules, for example. We can measure the frequency of executions of functions, if the program is suitably instrumented. We can also measure the frequency of executions of program operations. This facility already exists in the PASS software testing environment. The PMIP tool currently provides these data.

The objective of the DRAT is to provide a mechanism to compute the reliability of a complete software system and also to compute the reliability of the functionalities of the system. These values are determined from data obtained from an instrumented software system. The system is designed to provide a mechanism for continually updating the reliability estimate for the system.

If a system were fully instrumented to monitor transitions between software modules, the tool would be capable of providing dynamic reliability assessments of the instrumented system.

Equally important to the computation of the reliability are the confidence intervals for these reliability data. As more data are collected on the operation of a particular system, the confidence intervals should rapidly converge about the estimated reliability.

The basic operations of DRAT are to 1) create the initial transition matrix for all top-level program modules in PASS; 2) provide a mechanism to update this transition matrix as new data become available from the testing activity; and 3) provide a mechanism to delete or add modules to the transition matrix as they enter/leave the PASS software builds. While there are a total of 1,266 modules in the PASS system, only 769 of these are visible at the call level of program execution. The remainder of the program modules have essentially vanished from visibility because of the proprocessor include process. They exist only fully contained in other program modules.

The first step in using the reliability tool is to provide it with a complete list of all program modules. The cardinality of this set will be used by the tool to determine the size of the transition matrix. If there are 769 modules in a current build then the call matrix will have 771 rows and columns, including the FCOS and DR data rows and columns. The extra column will contain the DR count for each program. These DR counts will represent the transitions to the failure module represented internally as the last row in the call matrix. This last row will always contain zeros except for the last entry, which will always be a one.

The call matrix will be built for the system only once. After that, it will only be updated. There are two distinct sources of updates. First, we understand that new modules may enter at any time. Similarly, some existing modules may be phased out from time to time. Thus, the tool has a provision, before its execution, of permitting the transition matrix to be edited to 1) include new modules, 2) delete obsolete modules, and 3) update DR counts for the current working set of modules. This phase of the execution of the program is designed to change the static structure of the matrix.

The next phase of program execution is to update the contents of the transition matrix. Input to this phase of program execution will be a file containing PMIP output. Thus, as every test is completed, the new PMIP data may be used to update the contents of the call matrix.

To compute the reliability of the complete system, each row element will be divided by its row marginal to yield a transition matrix for reliability assessment.

The functional reliability of each test activity will be operationally defined by the contents of the PMIP file. That is, as each function is executed, a given subset of modules will be selected for execution by the function. The software will compute the *functional* reliability of the test just entered by partitioning the call matrix into a reduced matrix whose rows and columns are defined by the contents of the PMIP file. The reliability of each function will be computed from this reduced matrix.

Programmer Reference

The DRAT system has two basic modes of operation. It may be run interactively to update the module set representing the current state of the software system. As new program modules are added to the system they must also be added to the existing call matrix that resides on a file in the UNIX file system. As programs are removed from the build they must also be removed from the call matrix. As new DRs are recorded against existing modules, these new DR additions must be added to the information in the call matrix. These activities are all handled interactively.

When not in the interactive mode, the DRAT system will operate on new input as produced by the PMIP system through the supplied GET_TRANS filter. The purpose of the GET_TRANS filter is to reduce the PMIP data to a single file containing a set of records consisting of name pairs of program modules and a frequency of the associated transitions. The first name in the name pair is the name of the calling module. This name pair is followed by the relative # of times it occurred. The second name in the pair is the called module. If the DRAT tool is to be used outside of the Space Shuttle PASS environment, the calling pair data must be prepared through a user-supplied filter similar to GET_TRANS.

The basic operation of DRAT as a command line activity in UNIX is as follows:

```
drat -m module_file [-help] [-i] [-o out_file] [-c call_file]

            [-u call_file] [-t trans_file]
```

where the options and their associated files are defined as follows:

`-help`	print the option definitions and the above usage statement,
`-m module_file`	specifies file containing modules and DR data NOTE: -m option is always required
`-i`	enter interactive mode to add or delete modules

`-o out_file` specifies file in which to store transition matrix
NOTE: -o option required except in interactive mode

`-c call_file` create transition matrix (call_file is required)
NOTE: -c and -u options cannot be used together

`-u call_file` update transition matrix
NOTE: call_file and trans_file are both required

`-t trans_file` specifies the name of the file containing input transition matrix

The files named in the the command line are defined as follows:

1) `module_file ==>` line format: module-name DR-count

2) `out_file ==>` file format: module-column transition-matrix

3) `call_file ==>` line format: calling_module_name called_module_name frequency
This file is created from a PMIP output using the 'get_trans' filter.

4) `trans_file ==>` file format: module-column transition-matrix

The output on `stdout` from DRAT consists of two sets of numbers. The first number is the reliability of the system together with the $100(1-\alpha/2)\%$ confidence intervals for this estimate. The second set of numbers is the reliability of the functional set of program modules operationally defined by the PMIP data just processed together with the $100(1-\alpha/2)\%$ confidence intervals for this estimate.

References

Anger, F. D., J. C. Munson and R. V. Rodriguez (1994), "Temporal Complexity and Software Faults," *Proceedings of the IEEE International Symposium on Software Reliability Engineering 1994*, IEEE Computer Society Press, Los Alamitos, CA.

DeGroot, M. H. (1970), Optimal Statistical Decision, McGraw-Hill Book Company, New York.

Frankl, P. G, and E. J. Weyuker (1993), " Provable Improvements on Branch Testing," IEEE Transactions on Software Engineering, vol. SE-19 no. 10, pp. 962-976.

Halstead, M. H. (1977), Elements of Software Science. Elsevier, New York.

Jackson, J. E. (1991), A User's Guide to Principal Components. John Wiley and Sons, Inc., New York, New York.

Khoshgoftaar, T. M. and J. C. Munson (1992), "A Measure of Software System Complexity and Its Relationship to Faults," In Proceedings of the 1992 International Simulation Technology Conference, The Society for Computer Simulation, San Diego, CA, pp. 267-272.

Khoshgoftaar, T. M. and J. C. Munson (1990), "Predicting Software Development Errors Using Complexity Metrics," IEEE Journal on Selected Areas in Communications 8, pp.†253-261.

Lou, G., A. Das, and G. v. Bochmann (1994), "Software Testing Based on SDL Specifications With Save," IEEE Transactions on Software Engineering, vol. SE-20 no. 1, pp. 72-87.

McCabe, T. J. (1976), "A Complexity Measure," IEEE Transactions on Software Engineering, SE-2, pp. 308-320.

Mills, H. D. (1972), "On the Statistical Validation of Computer Programs," IBM Corporation Technical Report FSC-72-6015.

Morell, L. J. (1990), "A Theory of Fault-Based Testing," IEEE Transactions on Software Engineering, vol. 16, no. 8, pp. 844-857.

Munson, J. C. and T. M. Khoshgoftaar (1989), In "The Dimensionality of Program Complexity," Proceedings of the 11th Annual International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, pp. 245-253.

Munson, J. C. and T. M. Khoshgoftaar (1990), "Regression Modeling of Software Quality: An Empirical Investigation," Journal of Information and Software Technology, 32, pp. 105-114.

Munson, J. C. and T. M. Khoshgoftaar (1990) "Applications of a Relative Complexity Metric for Software Project Management," Journal of Systems and Software, 12, pp. 283-291.

Munson, J. C. and T. M. Khoshgoftaar (1990) "The Relative Software Complexity Metric: A Validation Study," In Proceedings of the Software Engineering 1990 Conference, Cambridge University Press, Cambridge, UK, pp. 89-102.

Munson, J. C. and T. M. Khoshgoftaar (1991), "The Use of Software Complexity Metrics in Software Reliability Modeling," In Proceedings of the IEEE International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, pp. 2-11.

Munson, J. and T. M. Khoshgoftaar (1992), "Measuring Dynamic Complexity," IEEE Software, pp. 48-55.

Munson, J. C. and T. M. Khoshgoftaar (1992), "The Detection of Fault-Prone Programs," IEEE Transactions on Software Engineering, SE-18, No. 5, pp. 423-433.

Munson, J. C. and R. H. Ravenel (1993), "Designing Reliable Software," Proceedings of the 1993 IEEE International Symposium of Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, pp. 45-54.

Musa, J. D. (1992), "The Operational Profile in Software Reliability Engineering: An Overview," In Proceedings of the IEEE International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, pp. 140-154.

Nakajo, T. and H. Kume (1991),"A Case History Analysis Of Software Error Cause-Effect Relationships," IEEE Transactions on Software Engineering, vol. SE-17, no. 8, pp. 830-838.

Raiffa, H. and R. Schlaifer (1961), Applied Statistical Decision Theory, Harvard University Press.

Weyuker, E. J. (1993), "More Experience With Data Flow Testing," IEEE Transactions on Software Engineering, vol. SE-19 no. 9, pp. 912-919.

Wilks, S. S. (1962), Mathematical Statistics, John Wiley and Sons, Inc., New York.

Appendix A

HALMet 3.1
Tools Package Installation Manual

HALMet / RELATIVE COMPLEXITY METRIC TOOLS PACKAGE--VERSION 3.1 INSTALLATION MANUAL

This manual provides the user with a detailed description of the requirements associated with the media transfer and subsequent installation of version 3.1 of the HALMet/RCM Tools package. The instructions contained herein should be explicitly followed with little or no need for changes that might normally be associated with differences in the Unix systems chosen for the installation.

For a detailed explanation of the operational procedures and use of this tools package, see the operation manual, Appendix B.

Media Transfer

There now exists a specific process by which efficient and dependable media transfer of the HALMet/ RCM tools package can be accomplished. This process is as follows:

1) "Tar" up the entire directory structure representing the distribution version of the HALMet tools package into a single tar file. Another tar file will be created containing the distribution version of the RCM tools package.

2) These tar files will then be compacted into much smaller binary files using the Unix 'compress' utility.

3) These files can then be downloaded via the standard File Transfer Protocol (ftp) that is associated with normal data transfer via the Internet.

Installation Requirements

After having downloaded the necessary tools package, it will then be necessary to perform an installation of the tools. In order to perform a full installation of version 3.1 of the HALMet HAL/S source code analyzer and version 2.0 of the PCA/RCM tools, the following requirements will have to be met:

1) Hardware: Sun SparcStation or comparable Unix-based workstation with at least 10 megabytes of available hard disk space and 5 megabytes of memory. These values represent the MINIMUM amount required for installation. An additional 10 megabytes of disk space may be necessary for operational purposes.

2) HALMet software: Version 3.1 of the HALMet analyzer as contained in the binary file, 'halmet31.tar.Z'.

3) PCA/RCM tools software: version 2.0 of the pca-rcm package as contained in the binary file, ‘pca-rcm.tar.Z’.

4) Standard Unix utilities:

uncompress	==>	restores compressed files to their original form
tar	==>	archives/unarchives complete directory structures
make	==>	compile and link manager and updater
cc	==>	"C" source code compiler
ld	==>	"C" object code linker
ar	==>	archiver for creating subroutine libraries
yacc	==>	parsing program generator
lex	==>	lexical analysis program generator

Installation Procedures

To install the packages:

1. Uncompress the files containing the ‘halmet’ HAL/S source code analyzer and the PCA/RCM tools package source code.

   > uncompress halmet31.tar.Z

   ==> produces a much larger file called halmet31.tar

   > uncompress pca-rcm.tar.Z

   ==> produces a larger file called pca-rcm.tar

2. Untar the resulting ‘tar’ files.

   > tar -xvf halmet31.tar

   ==> This will install version 3.1 of HALMet and its entire directory structure into a new directory called 'halmet3.1'.

   > tar -xvf pca-rcm.tar

   ==> This will install the source code for version 2.0 of PCA/RCM into a directory called pca-rcm.

3. Perform a 'make' in the ‘halmet’ library directory.

   > cd halmet3.1/lib

   > make

   ==> This will compile the library package.

4. Perform a make and installation in the ‘halmet’ source code directory.

   > cd halmet3.1/src

   > make install

      ==> This will create all the executables and place them into the 'halmet3.1/bin' directory.

5. Create the ‘PCA/RCM’ tool by performing a ‘make’ in its directory.

   > cd pca-rcm

   > make

      ==> This will create the ‘pca-rcm’ executable.

Sample Run

Also enclosed are both a small sample and a full Operational Increment of HAL/S source files that can be used to perform some test runs of the analyzer if so desired. To run the analyzer:

1. Add 'halmet3.1/bin' to your path.

The procedure for doing this is dependent on the shell being run. For example, in a c-shell (csh), the following might be used:

> setenv PATH "$PATH":$HOME/halmet3.1/bin

==> This appends the 'halmet3.1/bin' directory to the end of the PATH variable (assuming that the PATH variable exists and that the 'halmet3.1' directory exists in the user's HOME directory).

2. Execute the command to begin gathering the metrics. NOTE: Assuming that the provided sample is being used and can still be found as part of the original distribution, the following will be executed from within the 'halmet3.1' directory.

   > get_metrics -help

   ==> Displays a help message of available command-line options.

   > get_metrics -dir ./appl

   ==> This will run the analyzer and create a metrics file called metrics.txt'. (takes about 10-20 minutes to run.)

NOTE: If the same is run on the full Operational Increment, it will take approximately 17 hours to complete.

3. A sample data file has also been provide for a test run of the PCA/RCM tool and can be found in the ‘pca-rcm’ directory as ‘2301.data’. Therefore, the following command could be executed in the ‘pca-rcm’ directory:

   > pca-rcm

   ==> This will provide a usage statement and explanation of the options that must be specified on the command line.

   > pca-rcm 2301.data -base -d 2301.metrics -t 2301.trans

   ==> This will display a significant amout of information on the screen, so it might be desireable to pipe this to the ‘more’ command or redirect it to a file for later viewing. As per the operations manual, it will create two output files: 2301.metrics and 2301.trans.

Appendix B

HALMet 3.1
Tools Package Operation Manual

HALMet TOOLS PACKAGE
--VERSION 3.1--
OPERATION MANUAL

Overview

This manual provides the user with a detailed explanation of the use and processes associated with version 3.1 of the HALMet/RCM tools package. A step-by-step description outlining the significance and purpose of each of the individual executable functionalities will enable the user to better utilize the tools package and all of its capabilities. A full understanding of this manual is highly recommended but not necessarily required for implementation of the tools.

The following sections represent those sections that should be read as an absolute minimum before attempting to use the tools package: Overview, Introduction, get_metrics, rcm, join_metrics.

For a detailed description of the installation requirements and procedures for this tools package, see the INSTALLATION MANUAL.

As part of the standard installation, a set of documents will be installed that are suitable for using as manual pages for the unix 'man' command. They can be found in the 'doc' directory within the halmet3.1 directory and provide a detailed explanation of the proper use and options available to each of the primary operations performed by 'halmet'. A similar manual page exists for the PCA/RCM tool and can be found in the 'pca-rcm' directory.

Introduction

In general, the HALMet analysis tool is run with a single call to the primary controller or executor, 'get_metrics', which makes the necessary calls in the appropriate order to the rest of the executable programs. While the individual programs are kept in HALMet's 'bin' directory and can be accessed directly from there, most of them are only implemented through the use of the 'get_metrics' executor.

In order to perform a HALMet analysis run, the following three items must exist on the system: a correctly installed OI, the Major Action table, and the exceptions list. This is, of course, assuming that the necessary steps have been taken to properly and completely install the HALMet tools as specified in the Installation Manual.

A correctly installed OI requires that the OI be downloaded to the system on which HALMet is installed. It must then be set up such that it is composed of four directories (appl, incl80, ss, and mlib80), each of which contains the dataset members from the four original data sets (APPLSRC, INCL80, MLIB80, and SSSRC). The modules contained in these directories must be filtered in such a way as to remove anything

that is not HAL/S source code (i.e., line numbers). The line endings should also be translated to correspond to the system on which the OI is installed. Finally, appropriate suffixes must be attached to each module name that identify the type of the module. This module suffix mapping can be performed in accordance with the memberlist file that corresponds to the given OI.

The Major Action table is a file containing the module names and the number of major actions filed against each one as was previously extracted from an IMDB report. This file consists of one line for each module that is composed of the module name followed by white space followed by a single integer value representing the total number of major actions filed against that module. The name of this table defaults to 'maj_act_table'.

An exceptions list must also exist for proper operation of the HALMet tool, even if it is empty. This list is simply a file containing the module names, one module name per line, on which the HALMet analysis should not be run. While most of this need is taken care of by the suffix mapping, there may be some future need or reason for eliminating certain HAL/S modules from the analysis. Currently, this file is empty and defaults to the name 'x_list.txt'.

Previously Unidentified Metrics

Throughout this manual, many references are made to the various types of metrics that are being measured on the Hal/S source code by HALMet3.1, and specific names are given to each of these metrics. Most of the names used here are with direct respect to those metric names and their formal definitions described in Sections I and II.

The only metric values referenced in this manual that have not been previously identified in Sections I and II are the following:

Temporal metrics that are not significant to the current analysis:

Signal	==>	Number of SIGNAL statements in a program module
Termin	==>	Number of TERMINATE statements in a program module
MaxSiArg	==>	Largest number of arguments in any one SIGNAL statement
MaxTeArg	==>	Largest number of arguments in any one TERMINATE statement

Quality metrics:

Dis_Count	==>	Number of Discrepancy Reports in a program module Extracted from the prologues of each module (DR #'s)

Maj_Acts ==> Number of Major Actions for a program module
Accumulated from an IMDB report generated and transferred to us

Total_DRs ==> Summation of Dis_Count and Maj_Acts for a program module

CRs ==> Number of Change Requests in a program module
Extracted from the prologues of each module (CR #'s)

PCRs ==> Number of Program Change Requests for a program module
Extracted from the prologues of each module (PCR #'s)

Total_CRs ==> Summation of CRs and PCRs for a program module

Manual Organization

This manual is organized in such a way that each section of the manual corresponds to a specific executable functionality of the HALMet/RCM tools package. This executable functionality, in turn, can be identified and clearly related to the "C" source code on which it is based. This basis and a corresponding search can be performed throughout the manual with reference to those lines that contain the word, 'FILE:', beginning in the first column. The '.d' extensions indicate a relation between the "C" source files and the documentation as contained in the ‘doc’ directory of the full installation.

Each section is organized into the following subsections: FILE, DESCRIPTION, SYNOPSIS, INPUT, PROCESS, OUTPUT, COMPILATION, FILES, and SEE ALSO.

FILE:
get_metrics

DESCRIPTION:
The "get_metrics" executable is the top level metric analyzer. It is responsible for creating a metric dump of the HAL modules specified.

SYNOPSIS:
get_metrics -dir directory [-help] [-no_tree] [-no_def]

[-no_erase] [-t treefile] [-m modtable] [-x x_filename]

[-s struct_table] [-a maj_act_list]

[-out output_file] [-files file1 file2 ...]

-dir directory ==> Specifies the path to the APPLication directory. This option must be specified by the user. All HAL modules within the following directories are analyzed unless the -files option is used: appl/, incl80/, ss/, and mlib80/.

-help ==> Prints the usage line of get_metrics.

-no_tree ==> With this option enabled, the metric analyzer will skip the generation of the inclusion tree. This is typically used when a current tree file exists and is specified by the -t option.

-no_def ==> With this option given, the metric analyzer will skip the generation of the .def files.

-no_erase ==> This option tells the analyzer to allow the temporary files that are created to remain after the run.

-t treefile ==> Specifies the name of the treefile. If not specified, this value defaults to 'tree.out'.

-m modtable ==> Specifies the name of the module table. If not specified, this value defaults to 'modules.tab'.

-x x_filename ==> Specifies the name of the exceptions list. If not specified, this value defaults to 'x_list.txt'.

-s struct_table ==> Specifies the name of the structure table. If not specified, this value defaults to 'struct_table'.

-a maj_act_list ==> Specifies the name of the Major Actions list. If not specified, this value defaults to 'maj_act_table'.

-out output_file ==> Specifies the name of the metrics file. If not specified, this value defaults to 'metrics.txt'.

-files file1 file2 file3... ==> Instead of analyzing all HAL modules within the directories, this option will instruct the analyzer to limit the analysis to the given files. If this option is not given, all .HAL modules are examined.

INPUT:

The following files are used by the analyzer:

'*.HAL' ==> A HAL module which the user wishes to have analyzed.

'modtable' ==> The modules table containing a list of all HAL modules and a path to that module's file. This table is used to generate the inclusion tree.

'x_filename' ==> The exceptions list. Refer to exceptions.d for more details.

'maj_act_list' ==> The file containing a list of the number of major actions in a HAL module.

PROCESS:

"get_metrics" functions according to the following procedure:

1) Loads the exceptions list.

2) Loads the major actions table.

3) Runs "get_defs" which produces ".def" files for each file with a ".HAL" extension. The ".def" files include REPLACE statements and DEFINE directives at this point.

4) Runs "get_tree" which creates the inclusion tree.

5) On each source file, get_metrics will:

   a) Obtain the quality metrics from the module.

   b) Run "expand" to produce a dynamic state of the file or module by expanding its REPLACE and local DEFINE instances. Creates a ".exp" file.

   c) Run "count_tokens" to obtain the first half of the metrics. Creates a ".tok" file.

   d) Run "get_ctl_tokens" to extract the control flow tokens. Creates a ".ctl" file.

   e) Run "get_node_edges" to obtain the node pairs or edges. Creates a ".flo" file.

   f) Run "org_edges" to organize the node pairs. Creates a ".gph" file.

   g) Run "get_cyc_metrics" to obtain the remaining metrics. Creates a ".dat" file.

   h) Obtain the temporal metrics.

   i) Write the final metrics of the module to the output file.

   j) Remove all temporary files.

OUTPUT:

The final metrics file is in the following format with each field separated by white space:

MODULE-NAME 13-ORIGINAL-METS 10-TEMPORAL-METS 6-QUALITY-METS

where,

MODULE-NAME is the name of the HAL module,

13-ORIGINAL-METS is eta1 eta2 n1 n2 stmts loc comments nodes edges paths cycles max_path ave_path,

10-TEMPORAL-METS is

Sets Resets Signals Cancels Termins MaxSeAr MaxReAr MaxSiAr MaxCaAr MaxTeAr,

and

6-QUALITY-METS is dis_count maj_acts total_drs crs pcrs total_crs.

COMPILATION:

get_metrics is comprised of the following source files:

get_metrics : exceptions.c new_ls.c hal_dr.c halmet.c maj_act_table.c get_temporal.c libtree.a

FILES:

The following files are used or created by "get_metrics":

*.HAL

treefile

modtable

x_filename

struct_table

maj_act_list

output_file

*.def

*.exp

*.tok

*.ctl

*.flo

*.gph

*.dat

SEE ALSO:

exceptions.d

KNOWN LIMITATION:

None.

FILE:

exceptions

DESCRIPTION:

exceptions.c is a source file containing functions to access the exception list (typically named 'x_list.txt'). The exception list contains a list of filenames (HAL modules) which are to be excluded from analysis.

SYNOPSIS:

int get_exceptions(x_file)

    char *x_file;

get_exceptions() loads the exception list specified by x_file into memory.

void print_exceptions()

After the exception list has been loaded into memory, print_exceptions() will print the filenames within the exception list.

int is_exception(f_name)
    char *f_name;

After the exception list has been loaded into memory, is_exception will determine if the HAL module f_name is in the exception list.

INPUT:

get_exceptions() will read the specified file, char *x_list. x_list contains a list of filenames in the following format:

MODULE1.HAL

MODULE2.HAL

MODULE3.HAL

    :

MODULEn.HAL

PROCESS:

get_exceptions() puts into a structure each name in the exception list.

print_exceptions() traverses the structure printing each filename.

is_exception() traverses the list to determine if the specified filename is in the structure.

OUTPUT:

get_exceptions() returns the following conditions:

0 if an error occurred while trying to read the list

1 if the exception list has been properly loaded

print_exceptions() displays the list of filenames to stdout in the following format:

MODULE1.HAL

MODULE2.HAL

MODULE3.HAL

:

MODULEn.HAL

is_exceptions() returns the following conditions:

0 if the given file IS NOT in the exception list

1 if the given file IS in the exception list

COMPILATION:

The exceptions.c source is a file that compiles into "get_defs", "get_metrics", "expand", and "get_tree". The Make specification:

```
get_defs : repdef.o exceptions.o new_ls.o
get_metrics : exceptions.o new_ls.o hal_dr.o halmet.o
expand : proc_io.o replace.o exceptions.o
get_tree : get_include_pairs.o modtable.o exceptions.o new_ls.o
```

FILES:

x_list.txt ==> the exception list

SEE ALSO:

get_defs.d, get_metrics.d, expand.d, and get_tree.d

FILE: count_tokens

DESCRIPTION:

"count_tokens" is an executable responsible for obtaining the Halstead software science metric primitives from a HAL module. The following information in the metrics structure is obtained, as indicated by '==>':

```
        struct  metrics {

==>             char *module_name;      /* module name ==> first word in code */
==>             int     eta1,           /* unique operator count */
==>                     eta2,           /* unique operand count */
==>                     n1,             /* total operator count */
==>                     n2,             /* total operand count */
==>                     stmts,          /* total statement count */
==>                     loc,            /* total non-comment lines of code */
==>                     comments,       /* total comment count */
                        nodes,          /* total node count */
                        edges,          /* total edge count */
                        paths,          /* total path count */
                        cycles,         /* total cycle count */
                        max_path;       /* maximum path length */
                double   ave_path,      /* average path length */
                        data_struct;    /* to be dealt with later???? */
        } met;  /* software metrics to be accumulated */
```

SYNOPSIS:

count_tokens filename [-p]

filename specifies the expanded HAL module (.exp) to analyze.

-p option prints out a report of token counts. First, the tokens and their counts are displayed. Next, the identifiers and their counts are displayed.

INPUT:

"count_tokens" operates on an expanded HAL module only. The associated .HAL module needs to be in the same directory as the .exp module to obtain the lines of code and the lines of comment count.

PROCESS:

Refer to get_tokens.d, get_loc.d, and get_tok_funcs.d for more details.

OUTPUT:

The results are written to stdout in the following format:

```
printf("%-36.36s %d %d %d %d %d %d\n",
        met.module_name,
        met.eta1,
        met.eta2,
        met.n1,
        met.n2,
        met.stmts,
        met.loc,
        met.comments);
```

This format indicates that a character string containing the module name is printed followed by 7 numerical integer values containing the specified metrics in the order in which they are listed above followed by a new-line. The fields on each line are separated by white space.

COMPILATION:

"count_tokens" is comprised of get_tokens.c, get_tok_funcs.c, and get_loc.c. The Make specification:

count_tokens: get_loc.o get_tok_funcs.o get_tokens.o

FILES:

*.exp ==> the specified expanded file
*.HAL ==> the associated HAL module

SEE ALSO:

get_tokens.d, get_tok_funcs.d, get_loc.d, expand.d

FILE: get_tokens

DESCRIPTION:

The get_tokens.c source is a top level module responsible for obtaining the Halstead software science metrics from a HAL module.

This module validates the command line and communicates with get_tok_funcs.c and get_loc.c to determine the Halstead values of a HAL module.

SYNOPSIS:

```
main(argc, argv)
int argc;
char *argv[];
```

The argument count (argc) can range from two to three. argv specifies the name of the running program, the name of the .exp file to analyze, and an optional value to display the token counts.

INPUT:

get_tokens.c takes its input from the specified .exp file. The input from the file is read one line at a time and then separated into words.

word = [a-z]+[A-Z]+[0-9]+[_#%]

PROCESS:

This module passes the .HAL filename to get_loc.c to obtain the lines of code and lines of comment of the HAL file.

Next, the input words taken from the .exp file are passed to get_tok_funcs.c to obtain the operator and operand count of the HAL module. If a compiler directive is encountered (a line where the first character is a capital D), the line is skipped.

OUTPUT:

The results are written to stdout in the following manner:

```
printf("%-36.36s %d %d %d %d %d %d %d\n",
        met.module_name,
        met.eta1,
        met.eta2,
        met.n1,
        met.n2,
        met.stmts,
        met.loc,
        met.comments);
```

This format indicates that a character string containing the module name is printed followed by 7 numerical integer values containing the specified metrics in the order in which they are listed above followed by a new-line. The fields on each line are separated by white space.

COMPILATION:

The get_tokens.c source is a file that compiles into "count_tokens" along with get_loc.c and get_tok_funcs.c. The Make specification:

count_tokens: get_loc.o get_tok_funcs.o get_tokens.o
get_tokens.o: get_tokens.c

FILES:

*.exp ==> the specified expanded file
*.HAL ==> the associated HAL module of the .exp file

SEE ALSO:

count_tokens.d, get_tok_funcs.d, get_loc.d, expand

FILE: get_tok_funcs

DESCRIPTION:

get_tok_funcs.c is a module that analyzes the tokens passed to it. This module is used by count_tokens to count the number of operators and operands in a given HAL file.

This module is also used by get_ctl_tokens to examine the control flow tokens in a HAL file.

SYNOPSIS:

void Init_Op_Counts()

This function prepares the module for lexical analysis.

int Count_Token(word)
    char *word;

Count_Token() examines 'word' and determines if it is an operator or an operand. It then increments the appropriate count for word. The following members of the metrics structure are updated accordingly, as indicated by '==>':

```
    struct  metrics {

        char *module_name;      /* module name ==> first word in code */
```

```
==>             int     eta1,           /* unique operator count */
                        eta2,           /* unique operand count */
==>                     n1,             /* total operator count */
                        n2,             /* total operand count */
                        stmts,          /* total statement count */
                        loc,            /* total non-comment lines of code */
                        comments,       /* total comment count */
                        nodes,          /* total node count */
                        edges,          /* total edge count */
                        paths,          /* total path count */
                        cycles,         /* total cycle count */
                        max_path;       /* maximum path length */
                double  ave_path,       /* average path length */
                        data_struct;    /* to be dealt with later???? */
        } met;  /* software metrics to be accumulated */
```

void Get_Token_Totals()

After a HAL file has been completely analyzed, Get_Token_Totals() fills in the following metric values, as indicated by '==>':

```
        struct  metrics {

                char *module_name;      /* module name ==> first word in code */
                int     eta1,           /* unique operator count */
==>                     eta2,           /* unique operand count */
                        n1,             /* total operator count */
==>                     n2,             /* total operand count */
==>                     stmts,          /* total statement count */
                        loc,            /* total non-comment lines of code */
                        comments,       /* total comment count */
                        nodes,          /* total node count */
                        edges,          /* total edge count */
                        paths,          /* total path count */
                        cycles,         /* total cycle count */
                        max_path;       /* maximum path length */
                double  ave_path,       /* average path length */
                        data_struct;    /* to be dealt with later???? */
        } met;  /* software metrics to be accumulated */
```

void Print_Token_Cts()

This routine prints the results of the lexical analysis of a HAL module.

INPUT:

After this module has been initialized, Count_Token() receives its input as words. Words can be HAL/S reserved words or identifiers from the HAL module.

PROCESS:

After obtaining a word, the analyzer determines the token type: declarative, operator, or operand.

Declaratives are not counted as either operators or operands.

OUTPUT:

Output is facilitated with the Print_Token_Cts() function. A list of declaratives and operators with their associated totals is first displayed. Then, a list of identifiers, their count, and their type is displayed. The type value is as follows:

0 = scalar_type
1 = vector_type
2 = matrix_type
3 = other_type

COMPILATION:

The get_tok_funcs.c source is a file that compiles into "count_tokens" and "get_ctl_tokens". The Make specification:

count_tokens: get_tok_funcs.o get_loc.o get_tokens.o
get_ctl_tokens: get_tok_funcs.o proc_io.o get_ctl_flow.o
get_tok_funcs.o: get_tok_funcs.c t_y.tab.h

FILES:

none

SEE ALSO:

count_tokens.d, get_ctl_tokens.d, get_tokens.d, get_ctl_flow.d

FILE: get_loc

DESCRIPTION:

The get_loc.c source obtains the lines of code and lines of comment from a given HAL module.

SYNOPSIS:

int Get_LOC(filename)
char *filename;

The filename specifies the HAL module to analyze.

INPUT:

get_loc.c opens the specified HAL module and counts the number of lines of code and lines of comment.

A comment may be of C style or FORTRAN style.

A line of code is any nonblank line after the line has been stripped of comments.

PROCESS:

The given HAL module is opened and read in. If an error occurs in opening the file, an error message is reported and the function returns.

When a comment is encountered, it is stripped from the line and the total lines of comment is incremented. If the remaining line is nonblank, the total lines of code is incremented.

OUTPUT:

This function returns the following conditions:

0 ==> no errors
1 ==> error opening filename

The following members of the metrics structure are filled in:

met.loc
met.comments

COMPILATION:

The get_loc.c source is compiled into "count_tokens" along with get_tok_funcs.c and get_tokens.c. The Make specification:

count_tokens: get_loc.o get_tok_funcs.o get_tokens.o
get_loc.o: get_loc.c

FILES:

*.HAL ==> the specified HAL module filename

SEE ALSO:

count_tokens.d, get_tok_funcs.d, get_tokens.d

FILE: graphcvt --> org_edges

DESCRIPTION:

graphcvt.c compiles into "org_edges". This executable is responsible for converting a control flow graph represented as node pairs into a control flow graph represented as an adjacency list.

SYNOPSIS:

org_edges <filename>

<filename> (usually with the extension .flo) specifies an input file which contains the node pairs that are to be converted.

INPUT:

Input is taken from the specified file. The file contains information in the following format:

```
 1  2
 2  3
 :  :
%d %d
```

PROCESS:

The node pairs are obtained and represented internally as an adjacency list. After the node pairs have been exhausted, the list is dumped.

OUTPUT:

Results are written to standard output. Each line begins with a node. This node is then followed by the nodes to which it is connected. For example:

```
1    2
2    3
3    4    6
4    5
```

COMPILATION:

graphcvt.c compiles into the executable org_edges. The Make specification:

org_edges: graphcvt.o

FILES:

*.flo ==> the node pairs specification of a graph

FILE: join_metrics

DESCRIPTION:

join_metrics.c is a source file that compiles into the executable "join_metrics". "join_metrics" is responsible for joining the metrics file (created by "get_metrics") with the relative complexity and operational profiles. It creates a final file with all of the proper information for each module combined together.

SYNOPSIS:

join_metrics [-m metrics-file] [-r rel-comp-file]

[-p op_profile-file] [-x exceptions-file] output-filename

-m metrics-file ==> Specifies the metric dump that was created by "get_metrics".

-r rel-comp-file ==> Specifies the file containing the relative complexity of the modules.

-p op_profile-file ==> Specifies the file that contains the unsummed operational profiles.

-x exceptions-file ==> Specifies the exceptions list.

output-file ==> Specifies the output file to which to write.

INPUT:

"join_metrics" reads from the following files:

"metrics-file" is created by get_metrics and should be in the following format:

MODULE-NAME 13-ORIGINAL-METS 10-TEMPORAL-METS 6-QUALITY-METS

where,

MODULE-NAME is the name of the HAL module,

13-ORIGINAL-METS is
eta1 eta2 n1 n2 stmts loc comments nodes edges paths cycles max_path
ave_path,

10-TEMPORAL-METS is
Sets Resets Signals Cancels Termins MaxSeAr MaxReAr MaxSiAr MaxCaAr
MaxTeAr,

and

6-QUALITY-METS is
    dis_count maj_acts total_drs crs pcrs total_crs.

"rel-comp-file" holds the relative complexity of some of the HAL modules. It should be in the following format:

MOD-NAME FLOAT FLOAT FLOAT FLOAT REL-COMP.

"op_profile-file" holds the operational profiles. These may be unsummed. The file should be in the following format:

MOD-NAME FLOAT.

"exceptions-file" contains the exceptions list. Refer to exceptions.d.

PROCESS:

"join_metrics" operates according to the following procedure:

1) Loads the exceptions list.
2) Loads the metrics file.
3) Loads the relative complexity.
If a Relative Complexity exists for a module which is not in the metrics file, an error is reported.
4) Loads the operational profiles.
If an Operational Profile exists for a module which is not in the metrics file, an error message is reported.
5) Dumps the final metrics file to "output-file".

OUTPUT:

The "output-file" is in the following format with each field separated by white space:

MOD-NAME 13-ORIG-MET 6-TEMPORAL 6-QUALITY 1-RCM 1-OP 1-FCM

where,

6-TEMPORAL ==> Sets, Resets, Cancels, MaxSeAr, MaxReAr, MaxCaAr.

RCM ==> Relative Complexity Metric

OP ==> the summed Operation Profile

FCM ==> Functional Complexity Metric

Also, the last line of "output-file" has the total functional complexity.

COMPILATION:

join_metrics is comprised of two modules:

join_metrics : join_metrics.c exceptions.c

SEE ALSO:

exceptions.d, get_metrics.d

FILE: strip_zero

DESCRIPTION:

strip_zero.c is a source file that compiles into the executable strip_zero. Strip_zero is a simple filter that operates on metric dumps. Its purpose is to remove all COMPOOL files from the dump. Strip_zero should be used on files created by dump_zero to remove all zero level COMPOOLS.

SYNOPSIS:

strip_zero < <metric_dump> > <output file>

<metric_dump> is the metrics file that might contain COMPOOL files. <output file> is the newly created metrics file with COMPOOL files removed.

INPUT:

Strip_zero works on metric dumps. Typically, these are files created by get_metrics, dump_zero, or dump_summed. A metrics file is a text file with each line holding the metric values of a single HAL module. A line is in the following format:

<name> <value>...

where,

<name> is the filename of the HAL module, and
<value> is a certain metric value. The values are separated by spaces.

PROCESS:

Strip_zero takes the basename of <name> and tests if the first character is a 'C'. If this condition is false, the line is printed to stdout.

OUTPUT:

The output of a line is in the same format as its input.

COMPILATION:

The strip_zero.c source is a file that compiles into "strip_zero". The Make specification:
strip_zero : strip_zero.o

SEE ALSO:

dump_zero.d

FILE: dump_summed

DESCRIPTION:

"dump_summed" sums up all the zero level metrics and dumps the results (the level zero and level one modules) to an output file.

SYNOPSIS:

dump_summed <tree_file> <metrics_file> <output_file>

<tree_file> is the inclusion tree structure.
<metrics_file> is the metrics file to be summed.
<output_file> is where the resulting output will be saved.

INPUT:

Dump_summed works with two files: the tree structure and the metrics file. Both of these files are created by get_metrics.

Refer to read_tree.d for the format of the tree file.

The lines of the metrics file are formatted in the following style:

<name> <value>...

where,

<name> is the name of the module, and
<value> may be of type integer or float. Any number of values may follow the <name>, but this format must remain consistent with the other lines of the metrics file.

PROCESS:

After loading in the tree and metrics file, "dump_summed" will add to a parent node the values of its children. All of the metrics are then dumped to the output file.

OUTPUT:

The output file is in the same format as the metrics file.

COMPILATION:

The source file, dump_summed.c, compiles into "dump_summed". The Make specification:

dump_summed : dump_summed.c libtree.a

FILES:

tree_file ==> The inclusion tree. Typically created by get_metrics.
metrics_file ==> The metrics file. Typically created by metrics_file.

SEE ALSO:

get_metrics.d, read_tree.d

FILE: dump_zero

DESCRIPTION:

"dump_zero" sums up all of the zero level metrics and dumps the results (only the zero level modules) to an output file.

SYNOPSIS:

dump_zero <tree_file> <metrics_file> <output_file>

<tree_file> is the inclusion tree structure.
<metrics_file> is the metrics file to be summed.
<output_file> is where the resulting output will be saved.

INPUT:

Dump_zero works with two files: the tree structure and the metrics file. Both of these files are created by get_metrics.

Refer to read_tree.d for the format of the tree file.

The lines of the metrics file may be formatted in the following style:

<name> <value>...

where,

<name> is the name of the module, and
<value> may by of type integer or float. Any number of values may follow the <name>, but this format must remain consistent with the other lines of the metrics file.

PROCESS:

After loading in the tree and metrics file, "dump_zero" will add to a parent node the values of its children. The zero level values are then dumped to the output file.

OUTPUT:

The output file is in the same format as the metrics file.

COMPILATION:

The source file, dump_zero.c, compiles into "dump_zero". The Make specification:

dump_zero : dump_zero.c libtree.a

FILES:

tree_file ==> The inclusion tree. Typically created by get_metrics.
metrics_file ==> The metrics file. Typically created by metrics_file.

SEE ALSO:

get_metrics.d, read_tree.d

FILE: read_tree

DESCRIPTION:

"read_tree" is a utility program that will create a textual version of the inclusion tree. "read_tree" takes as input the treefile generated by "get_metrics" and writes to standard output the entire inclusion tree.

SYNOPSIS:

read_tree <treefile>
<treefile> is the a file created by "get_metrics"

INPUT:

"read_tree" takes as input the tree source file generated by a run of "get_metrics". The analyzer requires an inclusion tree in order to obtain accurate static analyses of the HAL source files. The "get_metrics" executive is used to generate a source file for the tree data structure.

The source file format appears below. Here, the parent file is not preceded by a greater-than sign (>) while its children are so preceded. So, below, the first file has three children, the second file (...CS4CPT.HAL) has no children, and the third file (...CS4DART.HAL) has 1 child.

```
H!appl/CS2PXT.HAL
>I!incl80/STRPXT.HAL
>C!appl/CS2PDT.HAL
>C!appl/CSAPDT.HAL
H!appl/CS4CPT.HAL
H!appl/CS4DART.HAL
>C!appl/CS4INB.HAL
```

PROCESS:

"read_tree" internally generates a tree structure from the source file. Duplicates on level zero only appear once within this internal representation. After the tree file has been completely loaded into memory, the tree structure is then traversed and the node names are written to standard output.

OUTPUT:

The expanded inclusion tree appears in textual form as follows:

```
H!0!appl/CS2PXT.HAL
I!1!incl80/STRPXT.HAL
C!1!appl/CS2PDT.HAL
C!1!appl/CSAPDT.HAL
H!0!appl/CS4CPT.HAL
H!0!appl/CS4DART.HAL
C!1!appl/CS4INB.HAL
```

The field separator is the exclamation point, so there are three fields. The first field specifies the DETERMINED (i.e., not given) type of the file. This single character field has three types as described below.

```
H ==> a HAL file
I ==> a file included directly by a HAL file        (this is also considered a HAL file)
C ==> a COMPOOL
```

The third field is obviously the path name of the file itself.

The second field describes the file's level in the current subtree. The basic or EXECUTIVE level is marked by a "0", indicating that the current file is the root of a tree and is a "sibling" to other root nodes or EXECUTIVE level modules. In general, a given entry with a level of k is the parent or ancestor of all files

immediately below it with a level greater than k. This means a subtree consists of a given entry having a level of k and all succeeding entries below it with a higher level.

COMPILATION:

"read_tree" code is based in read_tree.c and the tree library. The Make specification:

read_tree : read_tree.c libtree.a

FILES:

treefile ==> A tree source file generated by "get_metrics".

SEE ALSO:

get_metrics.d

FILE: read_subtree

DESCRIPTION:

"read_subtree" is a utility program that will print the children of a given node. This function can be used to find out what files a HAL module includes.

SYNOPSIS:

read_subtree <treefile> <filename>

<treefile> is the inclusion tree generated by "read_tree" and not the tree source file generated by "get_metrics".

<filename> is the name of the parent module.

INPUT:

"read_subtree" takes as input the textual version of the inclusion tree generated by "read_tree" and not the tree generated by "get_metrics".

Refer to read_tree.d for more details.

PROCESS:

"read_subtree" loads the inclusion tree into memory. The children of the given module name are then dumped to standard output.

OUTPUT:

The format of the subtree follows the output format of "read_tree". Refer to read_tree.d for more details.

COMPILATION:

"read_subtree" code is based on get_subtree.c and the tree library. The Make specification:

read_subtree : get_subtree.c libtree.a

FILES:

treefile ==> The inclusion tree file generated by "read_tree".

SEE ALSO:

read_tree.d

Appendix C

Principal Components Analysis/Relative Complexity Metric (PCA-RCM) 2.0 Tool Installation Manual

PRINCIPAL COMPONENTS ANALYSIS / RELATIVE COMPLEXITY METRIC TOOLS PACKAGE--VERSION 2.0 INSTALLATION MANUAL

OVERVIEW

This manual provides the user with a detailed description of the requirements associated with the media transfer and subsequent installation of version 2.0 of the PCA/RCM tools package. The instructions contained herein should be explicitly followed with little or no need for changes that might normally be associated with differences in the Unix systems chosen for the installation.

For a detailed explanation of the operational procedures and use of this tools package, see the OPERATION MANUAL, Appendix D.

MEDIA TRANSFER

There now exists a specific process for efficient and dependable media transfer of the HALMet/ RCM tools package:

1) "Tar" up the entire directory structure representing the distribution version of the PCA/RCM tools package into a single tar file.

2) This tar file will then be compacted into a much smaller binary file using the Unix 'compress' utility and made available for authorized transfer via standard File Transfer Protocol (ftp).

INSTALLATION REQUIREMENTS

After having downloaded the necessary tools package, it will then be necessary to perform an installation of the tools. In order to perform a full installation of version 2.0 of the PCA/RCM tools, the following requirements will have to be met:

1) Hardware: Sun SparcStation or comparable Unix-based workstation with at least 500 kilobytes of available hard disk space and 1 megabyte of memory. These values represent the MINIMUM amount required for installation and subsequent sample operations.

2) PCA/RCM tools software: version 2.0 of the PCA/RCM package as contained in the binary file, 'pca-rcm.tar.Z'.

3) Standard Unix utilities:

uncompress	==> restores compressed files to their original form
tar	==> archives/unarchives complete directory structures
make	==> compile and link manager and updater
cc	==> "C" source code compiler
ld	==> "C" object code linker

INSTALLATION PROCEDURES

To install the package:

1. Uncompress the file containing the PCA/RCM tools package source code.
    > uncompress pca-rcm.tar.Z
        ==> produces a larger file called pca-rcm.tar

2. Untar the resulting 'tar' file.
    > tar -xvf pca-rcm.tar
        ==> This will install the source code for version 2.0 of PCA/RCM
        into a directory called pca-rcm.

3. Create the 'PCA/RCM' tool by performing a 'make' in its directory.
    > cd pca-rcm
    > make ==> This will create the 'pca-rcm' executable.

SAMPLE RUN

A sample data file has been provide for a test run of the PCA/RCM tool and can be found in the 'pca-rcm' directory as '2301.data'. Therefore, the following commands could be executed in the 'pca-rcm' directory:
    > pca-rcm
        ==> This will provide a usage statement and explanation of the options that must be specified on the command line.
    > pca-rcm 2301.data -base -d 2301.metrics -t 2301.trans
        ==> This will display a significant amount of information on the screen, so it might be desirable to pipe this to the 'more' command or redirect it to a file for later viewing. As per the operations manual, it will create two output files: 2301.metrics and 2301.trans.

Appendix D

Principal Components Analysis/Relative Complexity Metric (PCA-RCM) 2.0 Tool Operation Manual

PRINCIPAL COMPONENTS ANALYSIS / RELATIVE COMPLEXITY METRIC TOOLS PACKAGE--VERSION 2.0 OPERATION MANUAL

FILE: pca-rcm

DESCRIPTION:

The "pca-rcm" executable is the final result of the compilation and linking of the source code representing the Principle Components Reduction tool and the Relative Complexity Metric tool. It is responsible for reducing a raw metrics data file into its related independent domains and then, using this reduction, it calculates the relative complexity for all of the modules occurring in the original data file.

SYNOPSIS:

pca-rcm <file1> -[base|build] -d <file2> -t <file3>

<file1> -- name of data file to be analyzed,
<file2> -- file in which to store domain and relative complexity metrics,
<file3> -- file in which to store (retrieve) the required info for baseline (build) analysis,
-base -- switch indicating a baseline analysis is desired,
-build -- switch indicating a build analysis is desired,
-d -- switch indicating that next argument is file2,
-t -- switch indicating that next argument is file3,

**** NOTES ****
1) Only 1 of -build or -base may be used at a time.
2) All options are required on the command line.
3) First argument given must be the data file to be analyzed.

Once created, a simple call to the "pca-rcm" executable with no arguments specified on the command line will display a help/usage statement with a listing and brief explanation of all of the options available. The user can then use the options to specify the input data file, the type of analysis desired, and the files in which to store the output.

INPUT:

The following files are used by the RCM tool:

<file1> ==> The data file that contains the raw metrics (i.e, that which was output by the HALMet analyzer). The only specifications governing the format of this file are:

1) The option must be given on the command line indicating whether the raw metrics file is to be analyzed as a baseline file or a build file.

A baseline analysis involves the complete reduction of the data file, calculation of the RCM, and the saving of the baseline data that will be necessary for subsequent build analyses. A build analysis involves the calculation of the RCM relative to a prior analysis that has been determined, previously, to be its baseline. The baseline analysis is, typically, performed once, while a build analysis will be performed a number of times.

See Section II for further explanation.

2) The lines of the data file must be composed of a single module name in the first column (or field) followed by the numerical values representing the raw metrics. These metric values can be represented as integers, floating point numbers, or in FORTRAN style scientific notation. The columns are separated by white space (spaces or tabs), and a carriage return indicates the end of the line.

Note that the columns and rows of this data file represent the variables and observations, respectively.

<file2> ==> The file that is specified on the command line with the '-d' switch which will serve as the output file for the final Domain Metrics and RCMs.

<file3> ==> The file that is specified on the command line with the '-t' switch in which will be stored certain baseline information during a 'baseline' analysis and from which will be read this baseline information during subsequent 'build' analyses. The information to be store in this file is necessary to make a 'build' analysis possible.

PROCESS:

"pca-rcm" functions according to the following procedure:

1) Opens the <file1> file and reads the first line to determine the number of variables that will be assumed to be the number of metrics taken on all of the modules in the data file. An error message is displayed if the file cannot be found.

2) If the analysis is to be a baseline analysis, the following is performed:

1. Reads each succeeding line of the <file1> file matching the length of each line (number of numerical values per line) to the length of the first metrics line.
2. Calculates column means, variance, covariance, and correlation.
3. Determines the independent domain metrics using a Principle Components Reduction of the correlation matrix that was created from the raw metrics data file.
4. Uses the domain metrics to calculate the Relative Complexity Metric for each module.
5. Stores the baseline information that will be required for related build analyses in the <file3> file.
6. Displays various levels of information from the resulting analysis on the screen (stdout), so that the user can monitor the progress of the analysis or redirect the information to a file.
7. Stores the module name, domain metrics, and Relative Complexity Metric in the <file2> file.

3) If the analysis is to be a build analysis, the following is performed:

1. Checks for the existence of the <file3> file and prints an error message if the file cannot be found.
2. Loads the information contained in the <file3> file.

3. Calculates the Relative Complexity for each module in the <file1> file relative to the baseline information.
4. Displays the messages from the resulting analysis on the screen (stdout).
5. Stores the module name, domain metrics, and Relative Complexity Metric in the <file2> file.

OUTPUT:

Error messages ==> various error messages are displayed as encountered.

Analysis output ==> various messages and data calculations are presented to the user for monitoring and progress evaluation throughout the full analysis of the input metrics data file.

Metrics output ==> a subset of the full analysis containing the module name, domain metrics, and Relative Complexity Metric is stored in the <file2> file according to the following format:

Module_Name Domain1 Domain2 .... DomainN RCM

Baseline data ==> that information from the output of a baseline analysis that is stored in the <file3> file so that it can later be used for a related build analysis, if one is performed.

COMPILATION:

"pca-rcm" is comprised of the following source file:
pca-rcm : pca-rcm.c

FILES:

The following files are used or created by "rcm":
<file1> ==> metrics input data file
<file2> ==> output of the domain metrics and relative complexity
<file3> ==> build analysis data related to the associated baseline
stdout ==> partial analysis output

SEE ALSO:

get_metrics.d, Section II

Appendix E

Dr Norman Schneidewind's Report: Experiment in Including Metrics in a Software Reliability Model

Appendix E

Work in Progress Report:
Experiment in Including Metrics in a Software Reliability Model

Dr. Norman F. Schneidewind
Naval Postgraduate School
Monterey, CA 93943

Objective

This paper describes an experiment to determine whether the inclusion of metrics in a reliability model that uses failure data for parameter estimation will improve its prediction accuracy. According to Loral Space Information Systems, formerly IBM Federal Systems Division, the Schneidewind Model fits "the data best" of the models that have been tried on the NASA Space Shuttle flight software. However, improvement in prediction accuracy that could be accomplished by any means, including the use of metrics, is welcome, given the safety critical nature of the application.

This experiment is not limited to the Space Shuttle application. Our objective also includes making a general contribution to the software measurement field regarding this very important and challenging issue: Will the inclusion of the characteristics of the software improve the accuracy of so-called "black box" reliability models? We suggest, for the reasons given below, that we should not pre-judge and assume that "obviously" the answer is in the affirmative. Although a specific model is used in the experiment, the results should at least be partially applicable to this class of model, in general, because all the models have the characteristic of using either failure count or time between failures data for parameter estimation.

Note: This experiment is not complete. This is a progress report. More software and different techniques for incorporating metrics in the model must be evaluated.

Purpose

1) If metrics improve prediction accuracy, they would be included in the model for future Space Shuttle software reliability prediction. 2) It is appealing to *believe* that high complexity results in low reliability; therefore, by this reasoning, the inclusion of software metrics should improve prediction accuracy.

We cover the following: revision of the software reliability model to include metrics; metrics validation methodology for producing validated metrics for classifying software quality; application of classification

results to represent the influence of software characteristics in the model; and some preliminary results comparing "with metrics" with "no metrics" predictions.

Background

Currently the model uses three parameters. Alpha (α) is the failure rate in the failure count interval *s-1*. (The interval *s-1* precedes the interval *s* where we start to use the failure data of the parameter estimation range s,t.) Beta (β) determines how fast the failure rate decreases. This model uses *s* to optimally select the failure data for parameter estimation [1,2,3]. In the model, the parameters α and β are assumed to be constant. Actually, it has been observed that they vary with execution time. There are three possible explanations for the fact that β varies with execution time, signifying a change in the rate of change in failure rate: 1) variations in the characteristics of the code at various execution times; 2) a reflection of the fact that, with fewer faults left in the code, as they are discovered and corrected, finding the remaining ones requires increasing amounts of execution time; or 3) noise. The last factor does not seem to be the case because the variation in the parameters is systematic and not random, as will be shown later. If the explanation is 1), the use of metrics in parameter estimation could be beneficial.

Theory

1. In general, modules that fail in execution have metrics values that are both larger and more correlated with discrepancy report counts (*drcount*), a measure of quality in the Shuttle, than in the population of modules (i.e., 1397 modules). Thus we can postulate a metrics Boolean discriminant function (BDF) that would classify quality with greater accuracy for failed modules than in the population of modules, of which the failed modules are a subset.

2. The failure rate per failure, beta (β), in the *revised* Schneidewind model (one that uses metrics) is a function of the initial failure rate alpha (α); execution time t; and $K=K_p/K_f$, where K_p is the *accuracy with which a BDF of metrics* D *can classify modules* M_p *in the population as having drcount*>0, *given that drcount*>0 and K_f is the *accuracy with which the same BDF can classify modules* M_f *as having drcount*>0, *given that these modules have been identified as causing failures*, where $M_f \subset M_p$. We assume K is approximately constant in the parameter estimation range s,t.

3. Beta can be modeled by equation (1), where the parameter estimation range of α and β is s,t. (In Method 2 of parameter estimation, s is the starting interval where failures will be counted.)

$$\beta=\alpha K^{(t-s+1)} \quad (1)$$

4. The rationale of this model of beta is as follows:

   a. The condition of the software is not static; it changes over time with modifications. Therefore, rather than β being constant, it should vary with execution time. Because small β is associated with a large failure rate and large β is associated with a small failure rate, β should increase for given values of K and t, as s increases, reflecting reliability growth over execution time (For parameter estimation purposes, $t \geq s$).

   b. We should have $\beta=\alpha$, when K=1, because this value of K corresponds to the situation on a *relative* basis, where the characteristics of failed software are no different than the characteristics of the software in the population. Also, when t=s-1 (the index value for the beginning of interval s), we should have $\beta=\alpha$, the initial failure rate.

5. We can substitute $\beta=\alpha K^{(t-s+1)}$ for β in the original Schneidewind model M.L.E. equations, obtain K from a BDF and substitute it in the M.L.E. equations, and estimate α. Once K and α have been obtained, we use them to estimate β. Once all parameters have been estimated, we can make a variety of predictions.

Analysis

1. To test the plausibility of $\beta=\alpha K^{(t-s+1)}$, we investigated whether $K=(\beta/\alpha)^{(1/(t-s+1))}$ is *approximately* constant in a subset of the parameter estimation range s,t (t=20) for three operational increments (OIs): OIB, OIC, and OID, where previously estimated paired values of β and α, corresponding to each value of s in its range, were used. The results are plotted in Figure 1, for OIC. In this analysis, K is approximately the ratio of $K=K_p/K_f$ we would have to obtain from the BDFs of the population and metrics data for these OIs, respectively, for the model to be validated. Although it might appear, because of the scale, that there are large variations in K, the range is actually quite narrow. The mean, standard deviation, and their ratio, for K are shown in the following Table 1.

Table 1: Values of K Versus Starting Interval (s)

Values of K (over s)	OIB	OIC	OID
Mean	.893	.841	.862
Stand. Dev.	.0349	.00967	.00423
Mean/Stand. Dev.	.0391	.0115	.00491

2. A second test of the plausibility of beta was to compute $\beta=\alpha K^{(t-s+1)}$, and compare it with the original beta that had been estimated using M.L.E. The purpose of this test was to evaluate how well the assumption of constant K would fare over the range of s. To do this, we used the mean of K, over s, in $\beta=\alpha K^{(t-s+1)}$. The plots of the original and new beta are shown in Figure 2 for OIC. Using a constant K did not produce too much variation between the old and new betas.

Revised M.L.E. Equations

1. To estimate the new alpha and beta and to incorporate K in the revised model, revised M.L.E. equations were produced. The original generalized likelihood function is shown in function (2) [1,2,3].

$$\log L = X_t[\log\ X_t 1\log(1\exp(\beta t))] + X_{s1}[\log(1\exp(-\beta(s1)))] + X_{s,t}[\log(1\exp(\beta))]\beta\sum_{k=0}^{ts}(s+k1)x_{s+k} \quad (2)$$

In this estimate: t is the last observed count interval; s is the index of time intervals; x_k is the number of observed failures in interval k; X_{s-1} is the number observed from 1 through s-1; $X_{s,t}$ is the number observed from s through t; and $X_t=X_{s-1}+X_{s,t}$.

Because software evolves over time and therefore the more recent failure data is more relevant with respect to the current and future process and product, only failure counts in the range s,t are used. This method had produced much more accurate predictions for the Shuttle than using all the failure data [1]. The following equation is used to estimate β.

$$\frac{1}{\exp(\beta)1}\frac{t-s+1}{\exp(\beta(t-s+1))1} = \sum_{k=0}^{ts} k\frac{x_{s+k}}{X_{s,t}} \quad (3)$$

The starting interval s is that value in the range 1,t that yields minimum mean square error (M.S.E.).

By substituting $\beta=\alpha K^{(t-s+1)}$ and $K_0=K^{(t-s+1)}$ in equation (2), taking $\partial(\log L)/\partial\alpha$ and setting the result to zero, and making the adjustments for estimating α and β in the range s,t, as was the case with equation (3), we arrive at equation (4) for estimating α in the revised M.L.E. Once α is estimated, β is obtained from $\beta=\alpha K^{(t-s+1)}$.

$$\frac{X_{s,t}}{\exp(\alpha K_0)1}\frac{t-s+1}{K_0\exp(\alpha K_0(t-s+1))} = \sum_{k=0}^{ts} k\,x_{s+k} \quad (4)$$

Revised Prediction Equations

1. Once the new parameters have been estimated, revised prediction equations are used to predict various reliability quantities for the Shuttle. For example, the old failure rate equation

$$f(s,t)=\alpha(\exp(-\beta(t-s+1))) \quad (5)$$

would be replaced by:

$$f(s,t,K)=\alpha(\exp(-\alpha K^{(t-s+1)}(t-s+1))). \quad (6)$$

Discriminative Power Validation Model

Now we explain how the *discriminative power validation model* is used to identify BDFs for obtaining K_p and K_f. The basis of this model is the validation of metric values that have the ability to discriminate high quality from low quality; these metric values are called critical values [4]. There are two types of criteria for validating critical values: statistical and application.

Critical Values

The Kolmogorov-Smirnov (K-S) test [6] is used to determine the critical value of a metric (M_{cj}). It tests whether the samples taken from different categories of data are from the same or different populations. Put differently, the method tests whether the sample cumulative distribution functions (CDFs) are from the same or different populations. The two populations are $F_i \leq F_c$ and $F_i > F_c$ (*drcount*=0 and *drcount*>0 in the *Shuttle*, respectively). In the K-S test, the test statistic is the maximum vertical distance between the CDFs of two samples. If the difference is significant, the value of M_{ij} corresponding to maximum CDF distance is used for M_{cj}, as expresses in equation (7).

$$K\text{-}S(M_{cj})=\max\{[CDF(M_{ij}/(F_i \leq F_c)]-[CDF(M_{ij}/(F_i > F_c)]\} \quad (7)$$

Metrics are added to the BDF (see equation (8)) in the order of their K-S distance.

Statistical Criteria

For the statistical criteria we use the *Contingency Table* (see **Table 2**) and its accompanying chi-square (χ^2) statistic [6]. In **Table 2**, M_{cj} and F_c classify modules into one of four categories. The left column contains modules where not one of the metrics exceeds its critical value; this condition is expressed with a Boolean *AND* function of the metrics. The right column contains modules where at least one metric exceeds its critical value; this condition is expressed by a Boolean *OR* function of the metrics. The top row contains modules that are high quality; these modules have a quality factor that does not exceed its critical

value (e.g., *drcount*=0). The bottom row contains modules that are low quality; these modules have a quality factor that exceeds its critical value (e.g., *drcount*>0).

Equation (8) gives the module count, based on Boolean functions of F_i and M_{ij}, that are calculated over the

$$C_{11} = \underset{i=1}{\overset{n}{COUNT}} \text{ FOR } ((F_i \text{ LE } F_c) \wedge (\text{MSUBi1LE } M_{c1}) \ldots \wedge (M_{ij} \text{ LE } M_{cj}) \ldots \wedge (M_{im} \text{ LE } M_{cm}))$$

$$C_{12} = \underset{i=1}{\overset{n}{COUNT}} \text{ FOR } ((F_i \text{ LE } F_c) \wedge ((M_{i1} > M_{c1}) \ldots \vee (M_{ij} > M_{cj}) \ldots \vee (M_{im} > M_{cm})))$$

$$C_{21} = \underset{i=1}{\overset{n}{COUNT}} \text{ FOR } ((F_i > F_c) \wedge (M_{i1} \text{ LE } M_{c1}) \ldots \wedge (M_{ij} \text{ LE } M_{cj}) \ldots \wedge (M_{im} \text{ LE } M_{cm}))$$

$$C_{22} = \underset{i=1}{\overset{n}{COUNT}} \text{ FOR } ((F_i > F_c) \wedge ((M_{i1} > M_{c1}) \ldots \vee (M_{ij} > M_{cj}) \ldots \vee (M_{im} > M_{cm})))$$

n modules for **m** metrics.

for j=1,...,m, and where COUNT(i)=COUNT(i-1)+1 FOR Boolean expression *true* and COUNT(i)=COUNT(i-1), otherwise; COUNT(0)=0.

The counts correspond to the cells of the *Contingency Table*, as shown in Table 2, where row and column totals are also shown. A special case of Table 2 is the use of a single metric. The analysis could also be generalized to include multiple quality factors, if necessary; in this case, the *Contingency Table* would have more than two rows.

Table 2: Validation Contingency Table

	$\wedge(M_{ij} \leq M_{cj})$	$\vee(M_{ij} > M_{cj})$	
$F_i \leq F_c$ High Quality	C_{11}	**Type 2** C_{12}	n_1
$F_i > F_c$ Low Quality	**Type 1** C_{21}	C_{22}	n_2
	N_1	N_2	n

We validate M_{cj} statistically by demonstrating that it partitions Table 2 in such a way that C_{11} and C_{22} are large relative to C_{12} and C_{21}. If this is the case, a large number of high-quality modules (e.g., modules with zero *drcount*) would have $M_{ij} \leq M_{cj}$ and would be correctly classified as high quality. Similarly, a large number of low-quality modules (e.g., modules with positive *drcount*) would have $M_{ij} > M_{cj}$ and would be correctly classified as low quality. The degree to which this is the case is estimated by the chi-square (χ^2) statistic. If calculated $\chi^2_c > \chi^2_s$ (chi-square at specified α_s) and if calculated $\alpha_c < \alpha_s$, we conclude that M_{cj} is statistically significant.

Application Criteria

This validation is treated under the categories of **Misclassification**, **Inspection**, and **Quality** [5].

Misclassification

We estimate the *discriminative power* of M_{cj} by noting in Table 2 that ideally $C_{11}=n_1=N_1$, $C_{12}=0$, $C_{21}=0$, $C_{22}=n_2=N_2$. The extent that this is not the case is estimated by the number of *Type 1* misclassifications (i.e., the module has *low quality* and the metrics "say" it has *high quality*) and by the number of *Type 2* misclassifications (i.e., the module has *high quality* and the metrics "say" it has *low quality*). Thus we define the following measures of misclassification:

Proportion of Type 1: $P_1=C_{21}/n$ (9)

Proportion of Type 2: $P_2=C_{12}/n$ (10)

LQC: Proportion of low quality (i.e., *drcount*>0) software correctly classified=C_{22}/n_2 (11)

$$RF = \sum_{i=1}^{n} F_i \text{ FOR } ((M_{i1} \text{ LEMSUBc1}) \ldots \wedge (M_{ij} \text{ LE } M_{cj}) \ldots \wedge (M_{im} \text{ LE } M_{cm})) \textit{ for } j = 1,\ldots,m. \quad (12)$$

Quality

We estimate *discriminative power* by summing remaining quality factor RF (e.g., remaining *drcount*), given by equation (12). This is the sum of F_i not caught by inspection because ($M_{ij} \leq M_{cj}$) for these modules.

We estimate the proportion remaining by equation (13), where TF is the total F_i before inspection.

RFP=RF/TF (13)

In addition, we estimate the count of modules remaining that have $F_i>0$. The proportion remaining RMP is given by equation (14). Note that RMP=P_1 (proportion of *Type 1* misclassifications) when $F_c=0$ (i.e., the only modules with $F_i>0$ will be in the C_{21} cell); see Table 2.

$$RMP=(C_{11}+C_{21})/n, \text{ FOR } F_i>0 \quad (14)$$

Inspection

Inspection is one of the costs of high quality. We are interested in weighing inspection requirements against the quality that is achieved for various values of M_{cj}. This is another way to estimate *discriminative power*. We estimate inspection requirements by noting that all modules with $M_{ij}>M_{cj}$ must be inspected; this is the count $C_{12}+C_{22}$. Thus the proportion of modules that must be inspected is given by:

$$I=(C_{12}+C_{22})/n \quad (15)$$

Validation Example

An example of validating BDFs in the population using the above equations is shown in Table 3. The critical values of the metrics and the order of adding metrics was determined by applying equation (7), which yielded the K-S distance values in the table.

Stopping Rules for Adding Metrics

One rule for stopping the addition of metrics in a BDF is to quit when RFP no longer decreases as metrics are added. This is the *maximum quality* rule. The last two rows of Table 3 illustrate this rule where only 1.24 percent of the *drcount* is not caught by inspection. The addition of the sixth metric *nodes* has not improved quality. If it is important to strike a balance between quality and cost (i.e., between RF and I), we add metrics until the ratio of the relative change in RF to the relative change in I is maximum, as given by the *quality inspection ratio* in equation (16), where *i* refers to the previous RFP and I:

$$QIR=(|\Delta RFP|/\Delta I)(I_i/RFP_i) \quad (16)$$

The third row in Table 4 has the highest value (4.82). So, by this criterion, three metrics would be used.

This analysis indicates the importance of performing a *marginal analysis*, which involves adding the metrics one at a time to the BDF and observing the effect on quality and inspection cost. If, on the other hand, many metrics are added at once, the contribution of individual metrics is obscured.

Table 3: *Discriminative Power* Validity Evaluation (Population)

Metrics	C_c	S_c	$E2_c$	L_c	$E1_c$	N_c	P_1 %	P_2 %	LQC %	RFP %	RMP %	I %	χ^2_c
C	63						6.22	15.10	84.90	6.13	6.23	50.1	472.10
C,S	63	27					3.22	22.12	92.19	2.95	3.22	60.1	417.90
C,S,E2	63	27	45				2.51	24.62	93.92	2.17	2.51	63.4	392.35
C,S,E2,L	63	27	45	29			2.00	29.35	95.14	1.78	2.00	68.6	318.83
C,S,E2,L,E1	63	27	45	29	9		1.43	34.93	96.53	1.24	1.43	74.7	244.62
C,S,E2,L,E1,N	63	27	45	29	9	17	1.43	34.93	96.53	1.24	1.43	74.7	244.62
K-S Distance	.59	.51	.47	.46	.43	.43							

C: total comment count

S: total statement count (executable code; no comments)

E2: unique operand count

L: total non-commented lines of code

E1: unique operator count

N : total node count (in control graph)

C_c: critical value of comments

S_c : critical value of statements

$E2_c$: critical value of unique operator count

L_c : critical value of lines of code

$E1_c$: critical value of unique operator count

N_c : critical value of *nodes*

P_1: Percentage of Type 1 misclassifications

P_2: Percentage of Type 2 misclassifications

LQC: Percentage of low quality software correctly classified

RFP: Percentage of *drcount* remaining after inspection

RMP: Percentage of modules remaining after inspection with *drcount*>0

I: Percentage of modules inspected

χ^2_c: Calculated chi-square; α_c=0 in all cases.

Table 4: *Discriminative Power* Marginal Analysis (Population)

Metrics	QIR
C	-
C,S	2.60
C,S,E2	4.82
C,S,E2,L	2.19
C,S,E2,L,E1	3.41
C,S,E2,L,E1,N	-

QIR: Quality Inspection Ratio

Some Preliminary Results

We now apply the metrics that were validated in the previous section to computing $K=K_p/K_f$. Using the first BDF in Table 3, the one consisting only of *comments*, it correctly classifies every module in Table 5, which consists of modules that have caused failures in *OIC*. Therefore, $K_f=1$. To obtain K_p, we note that LQC=.849 in Table 3 for the first BDF. Therefore $K=K_p/K_f$ =.849/1.0=.849. Using the revised M.L.E. equation (4), we estimate α and then β from equation (1). Then we predict cumulative failures at T=30, using equation (17) for both the "no metrics" and "with metrics" cases, and compare the two in Figure 3. We also note the pronounced difference between the means of metrics in the sample (means of the metrics associated with failed modules) and the means of metrics in the population.

In addition, we compute the M.S.E. for the two cases with equation (18) and compare the results in Table 6. Both Figure 3 and Table 6 indicate the "with metrics" prediction is more accurate.

$$F(T)=(\alpha/\beta)[1-\exp(-\beta((T-s+1)))]+X_{s-1} \quad (17)$$

Currently failure data from additional OIs are being investigated. Complete results are not yet available.

$$MSE_F = \frac{\sum_{i=s}^{t} [\alpha / \beta(1\exp(\beta(is+1)))\, X_{s,i}]^2}{ts+1} \qquad (18)$$

Table 5: Failure and Metrics Data for OIC

Failure Number	Severity Level	Module ID	comments	stmts	eta2	loc	eta1	nodes	dr cnt
1	2	13	493	738	336	1026	46	394	22
2	3	974	299	192	141	240	31	98	2
3	2	1286	115	110	104	180	28	48	5
4	3	711	205	*1*	*2*	222	*5*	96	6
5	3	1300	82	*3*	*5*	53	*8*	20	1
6	3	515	851	875	338	1101	44	529	15
7	2	464	69	*15*	*20*	62	16	*12*	4
7	2	465	76	30	*45*	86	24	21	4
7	2	466	68	*15*	*20*	61	16	*12*	4
7	2	467	72	30	*45*	77	24	21	2
7	2	468	153	*10*	*29*	145	11	75	3
7	2	472	100	*1*	*4*	107	*6*	40	1
8	4	555	943	819	432	947	34	174	26
10	3	904	122	128	86	163	31	64	1
11	4	882	157	107	91	129	30	51	5
Critical Value			63	27	45	29	9	17	0
Sample Mean			253.7	204.9	113.2	306.6	23.6	110.3	6.7
Population Mean			134.6	70.2	81.3	132.3	16.7	28.4	1.8

Metric values in *italics* would fail to flag modules identified with failures (i.e., value ≤critical value).
Difference between Sample Mean and Population Mean significant at $\alpha<.05$ except for eta2.

Table 6: Prediction "With Metrics" versus "No Metrics"

	s	α	β	F(30)	MSE_F
With Metrics	7	1.3530	.13681	11.53	.356
No Metrics	7	1.3667	.12574	12.34	.505

Actual cumulative failures at T=30 is 12

F(30): Predicted cumulative failures at T=30

$MSE_{F:}$ Mean Square Error in range 7,30

Summary

A general framework for including metrics in the *Schneidewind Software Reliability Model* has been developed, which may be applicable to other models. Because the appearance of the factor K in the parameter β is systematic rather than random, it was hypothesized that K accounts for the influence of various software characteristics, as represented by the metrics, on the occurrence of failures. The process of using BDFs to provide the factor K in the revised model was illustrated. Many more OIs and modules must be investigated before any conclusions can be drawn about whether the inclusion of metrics in the model improves prediction accuracy. In addition to the method for obtaining K that has been described, the following methods will be explored:

1. Compute K as the ratio of successful classifications to total classifications, considering the metrics individually, for each module in the failure data, obtaining n values of K per OI, where n is the number of modules that caused failures. Estimate K as a function of M_{ij} in the failure data.

2. Compute the mean of K as the ratio of total successful classifications to total classifications by m metrics across n modules, considering the metrics individually in the failure data, obtaining one value of K per OI. Estimate the mean of K as a function of the mean of M_{ij}.

   With methods 1 and 2, metrics would be included *directly* in the model.

3. Obtain the coefficients of a multivariate discriminant function in the population. Using the means of the metrics in the population compute its discriminant function. Do the same with the means of the failed modules. Set K equal to the ratio of the former by the latter.

4. Forget metrics and estimate K by M.L.E from equation (4) *directly*!

Acknowledgments

We wish to acknowledge the support provided for this project by Dr. William Farr, Naval Surface Warfare Center, United States Marine Corps, and Ms. Alice Lee of NASA; the data provided by Prof. John Munson

of the University of Idaho and the metrics analyzer he developed for the HAL/S language; and the assistance provided by Mr. Ted Keller and Ms. Patti Thornton, of LORAL Space Information Systems.

References

[1] Recommended Practice for Software Reliability, R-013-1992, American National Standards Institute/American Institute of Aeronautics and Astronautics, 370 L'Enfant Promenade, SW, Washington, DC 20024, 1993.

[2] William H. Farr and Oliver D. Smith, Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide, NSWCDD TR-84-373, Revision 3, Naval Surface Weapons Center, Dahlgren Division, Revised September 1993.

[3] Norman F. Schneidewind, "Software Reliability Model with Optimal Selection of Failure Data," IEEE Transactions on Software Engineering, Vol. 19, No. 11, November 1993, pp. 1095-1104.

[4] Norman F. Schneidewind, "Methodology for Validating Software Metrics," IEEE Transactions on Software Engineering, Vol. 18, No. 5, May 1992, pp. 410-422.

[5] Norman F. Schneidewind, "Validating Metrics for Controlling and Predicting the Quality of Space Shuttle Flight Software," IEEE Computer, Vol. 27, No. 8, August, 1994, pp. 50-57.

[6] W.J. Conover, Practical Nonparametric Statistics, John Wiley & Sons, Inc., New York, NY., 1971.

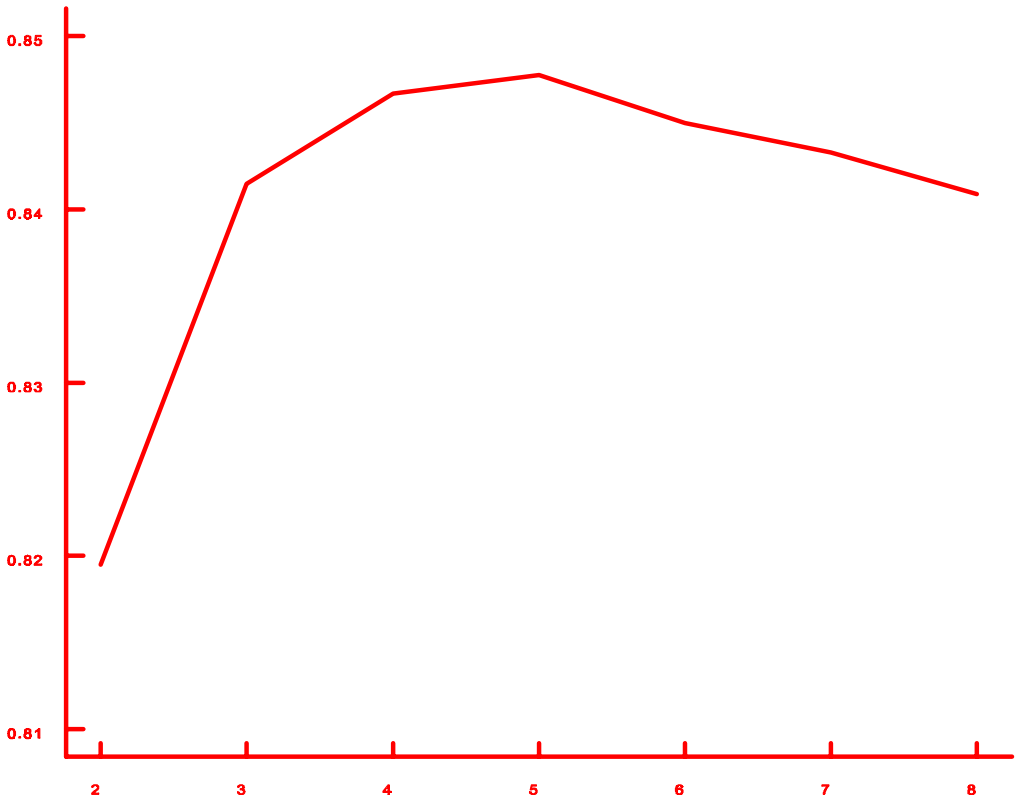
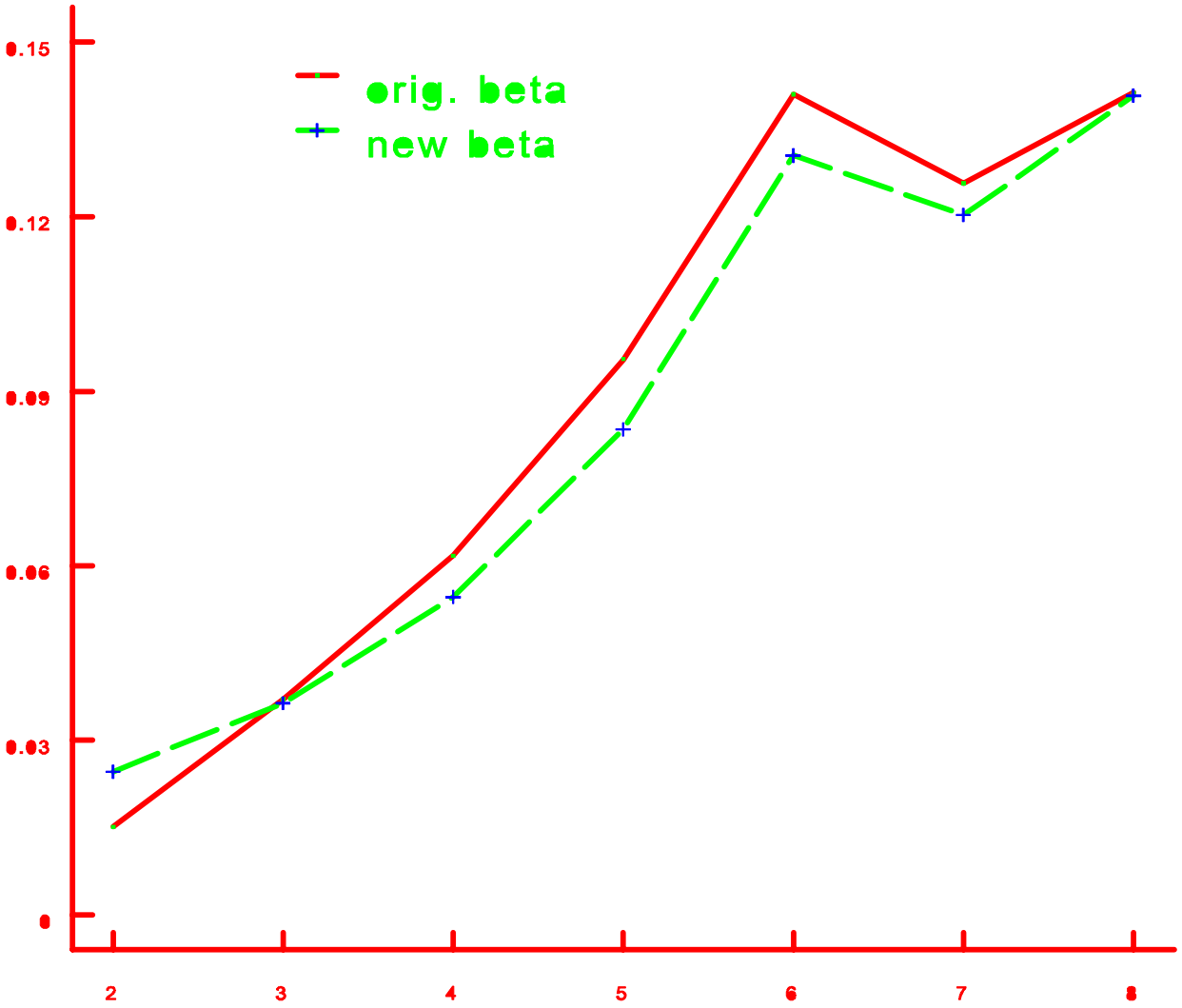

0.85
0.84
0.83
0.82
0.81
2
3
4
5
6
7
8
Starting Interval(s)
K versus Starting Interval s, OIC, t=20

Figure 1

original beta: from MLE
new beta=(alpha)*(K**(t-s+1)

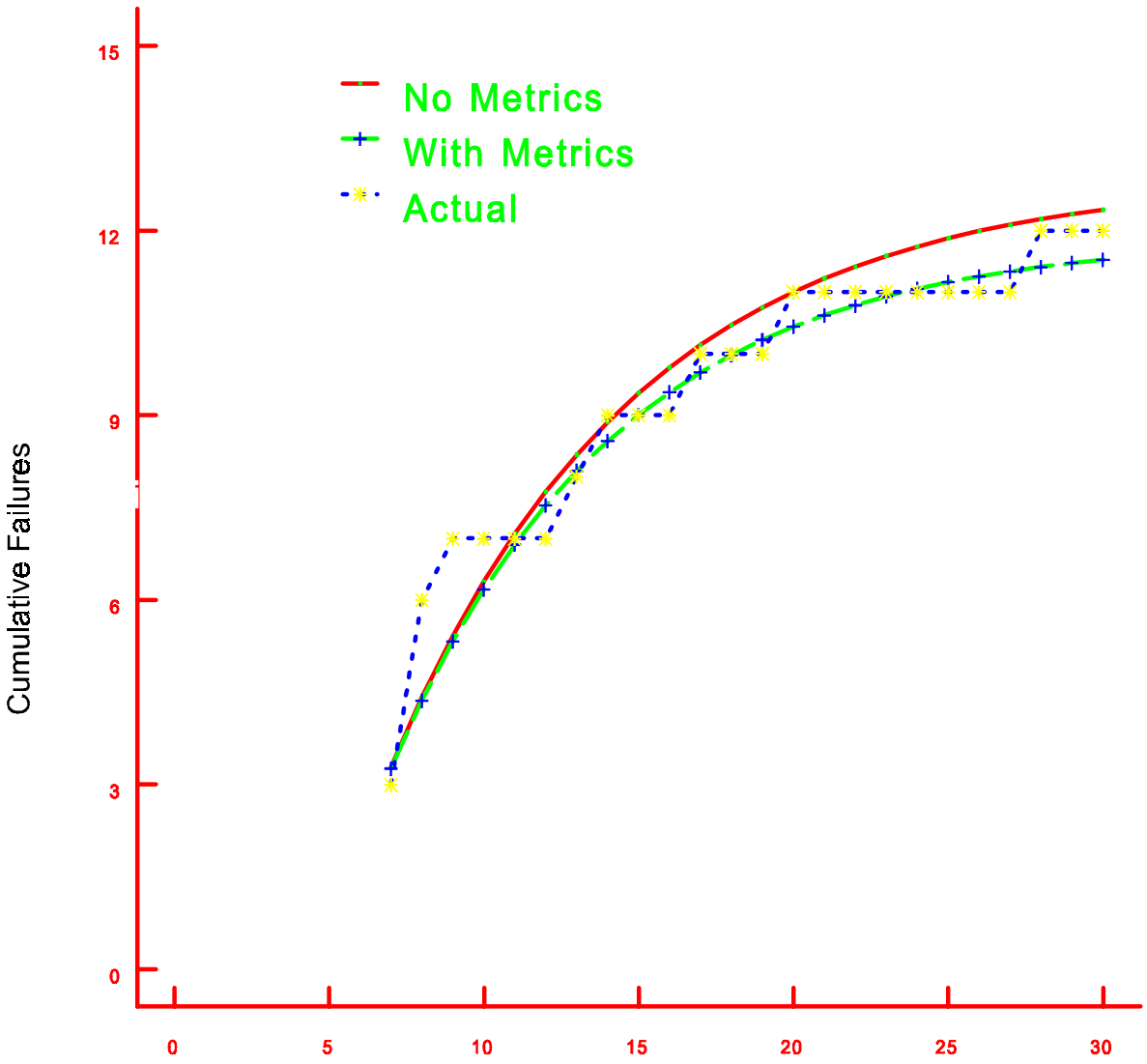


Starting Interval(s)

original beta & new beta vs s, OI5, t=20

Figure 2

Predicted failures: no metrics and with metrics. Actual failures.



Cumulative failures for OIC

Figure 3